**1D, 2D, MultiDimensional Array – Assignments**

**1D Array**

**1. Refer the code snippet and answer the queries**

int main()

{

int array[100];

int \*ptr;

// do something

}

**Q1: Can pointer be used in Array-style syntax? e.g. ptr[10], ptr[0]**

Yes, a pointer can be used in array-style syntax. Specifically, ptr[10] is equivalent to \*(ptr + 10), and ptr[0] is equivalent to \*(ptr + 0), which dereferences the pointer at the offset specified.

So, if ptr points to the first element of an array (or a valid memory location), you can use array-style indexing to access the elements, as long as the pointer has been properly initialized to point to a valid location.

**Q2: Can Array be used in Pointer-style syntax? e.g. \*array, \*(array + 0), \*(array + 10)**

Yes, an array can be used in pointer-style syntax. In fact, in C, the name of an array (array) is equivalent to a pointer to its first element. For example:

* \*array is equivalent to array[0] (dereferencing the pointer to get the first element).
* \*(array + 0) is also equivalent to array[0] (dereferencing the pointer after adding 0).
* \*(array + 10) is equivalent to array[10] (dereferencing the pointer after adding an offset of 10).

So, the array name itself can be treated as a pointer to the first element, and pointer arithmetic can be used to access its elements.

**Q3: is ptr++ valid?**

Yes, ptr++ is valid as long as ptr is a valid pointer. It increments the pointer ptr so that it points to the next element in the array (or the next memory location if ptr is pointing to dynamic memory or a valid memory range). If ptr points to an array, ptr++ will increment the pointer by the size of the type it points to (e.g., 4 bytes if it's a pointer to int).

However, for ptr++ to be valid, ptr must have been properly initialized (e.g., it should point to a valid memory location).

**Q4: is array++ valid?**

No, array++ is not valid. In C, the name of an array (array) is a constant pointer to the first element of the array, and you cannot modify it (i.e., increment it). Attempting to use array++ would result in a compilation error because array is not a modifiable value.

While you can use pointer arithmetic with array (like \*(array + 10)), you cannot modify the array name itself (i.e., you cannot do array++ or array += 1).

**Q5: what is sizeof(array)?**

sizeof(array) will give the total size of the entire array in bytes. Since array is declared as int array[100];, and assuming sizeof(int) is 4 bytes (which is common on many systems), the result of sizeof(array) would be:

sizeof(array) = 100 \* sizeof(int) = 100 \* 4 = 400 bytes

So, sizeof(array) would return 400 bytes.

**Q6: what is sizeof(ptr)?**

sizeof(ptr) will give the size of the pointer variable ptr. In this case, ptr is of type int\*, which means it's a pointer to an int.

The size of a pointer (int\*) is typically 4 bytes on a 32-bit system or 8 bytes on a 64-bit system. However, the size of a pointer depends on the architecture of the machine, not the type it points to.

So, for most modern systems (e.g., 64-bit systems), sizeof(ptr) would typically return 8 bytes, while on a 32-bit system, it would likely return 4 bytes.

**2.)Refer the code snippet below. Comment on the other elements (other than those that are explicitly initialized) of all array variables in code snippet below.**

#define MAX 100

int main()

{

int arr[MAX] = {11,22,33};

int arr1[MAX]={0};

static int arr2[MAX];

}

Array arr[MAX] = {11, 22, 33};

* This array is declared with 100 elements (since MAX is defined as 100), and only the first three elements are explicitly initialized (11, 22, and 33).
* Explicit initialization:
  + arr[0] = 11
  + arr[1] = 22
  + arr[2] = 33
* Uninitialized elements:
  + All remaining elements (arr[3] to arr[99]) are implicitly initialized to 0. This is because when an array is partially initialized, the uninitialized elements are automatically set to 0 in C, as per the C standard.

So, after initialization:

* arr[0] = 11
* arr[1] = 22
* arr[2] = 33
* arr[3] to arr[99] = 0

2. Array arr1[MAX] = {0};

* This array is also declared with 100 elements, and the first element is explicitly initialized to 0. This initialization is done using {0}, which means the first element gets the value 0, and all the remaining elements (because of the default behavior) are also initialized to 0.
* Explicit initialization:
  + arr1[0] = 0
* Uninitialized elements:

Since only arr1[0] is initialized and no other value is provided, all the remaining elements (arr1[1] to arr1[99]) will also be initialized to 0.

So, after initialization:

* arr1[0] = 0
* arr1[1] to arr1[99] = 0

3. Array static int arr2[MAX];

* This array is declared as static. In C, static variables are automatically initialized to zero if they are not explicitly initialized. This applies to both global and local static variables. The key difference with non-static variables is that static variables retain their value between function calls, but initialization rules remain the same.
* Explicit initialization:
  + No explicit initialization is provided, so the array is initialized to zero automatically by the compiler.
* Uninitialized elements:
  + All elements of arr2 will be automatically initialized to 0, due to the behavior of static variables in C.

So, after initialization:

* arr2[0] to arr2[99] = 0

**Summary of Array Initialization:**

1. **arr[MAX] = {11, 22, 33};**:
   * arr[0] = 11
   * arr[1] = 22
   * arr[2] = 33
   * arr[3] to arr[99] = 0 (implicitly initialized)
2. **arr1[MAX] = {0};**:
   * arr1[0] = 0
   * arr1[1] to arr1[99] = 0 (implicitly initialized)
3. **static int arr2[MAX];**:
   * arr2[0] to arr2[99] = 0 (automatically initialized to 0 due to static)

**Key Observations:**

* **Uninitialized elements:** For non-static arrays like arr and arr1, the C language initializes uninitialized elements to zero if the array is partially initialized.
* **Static arrays:** For static arrays (arr2), all elements are automatically initialized to zero, even if you don’t explicitly initialize them.

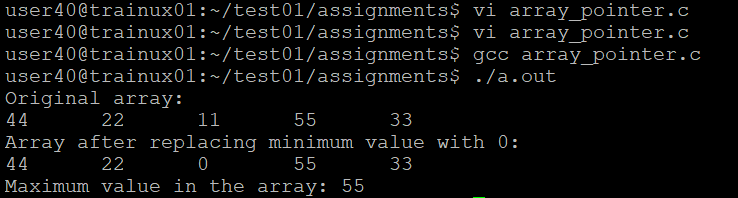
**3. Refer the program “array\_pointer.c”. Add a function getmax() to find the maximum in the array and call in main() and display the result.**

A screenshot of a computer program

Description automatically generated

A computer screen with text

Description automatically generated



**4. Extend the code given below to read N and a start value from the user to perform the given operations.**

**#define MAX 100**

**int main()**

**{**

**int arr[MAX] = {11,22,33};**

**}**

**Add the following functions choosing proper input, output and return.**

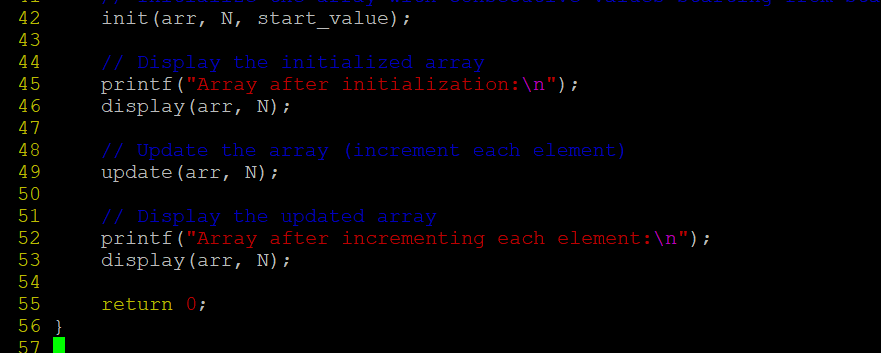
**a. init() - Use the inputs to initialize the first N elements of the array with N consequetive values starting with given start value .**

**b. update() – increment value of every element in the array**

**c. display() – display the contents of array**

A screenshot of a computer program

Description automatically generated



A screen shot of a computer

Description automatically generated

**2D, MultiDimensional Arrays**

**1. Implement sort() to sort a given array. Refer the code snippet below.**

**int main()**

**{**

**char arr[]= “xaybz”;**

**sort(arr, sizeof(arr)/sizeof(arr[0]);**

**return 0;**

**}**

A computer screen shot of text

Description automatically generated

A screen shot of a computer

Description automatically generated

**2. Refer the code snippet below.**

**int main()**

**{**

**char arr[][3] = {**

**sort(arr, sizeof(arr)/sizeof(arr[0]);**

**return 0;**

**}**

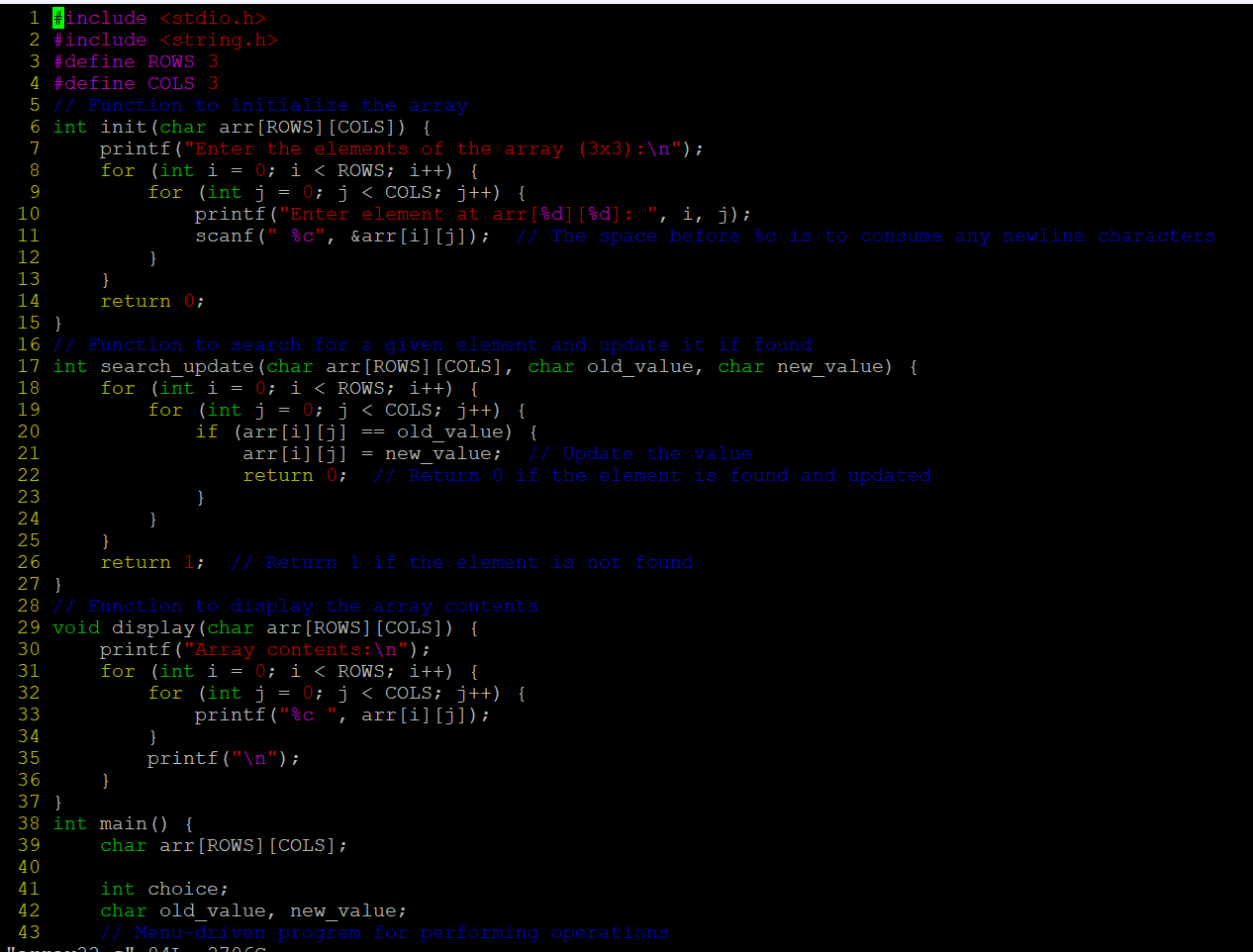
**Allow user to perform the following operations.**

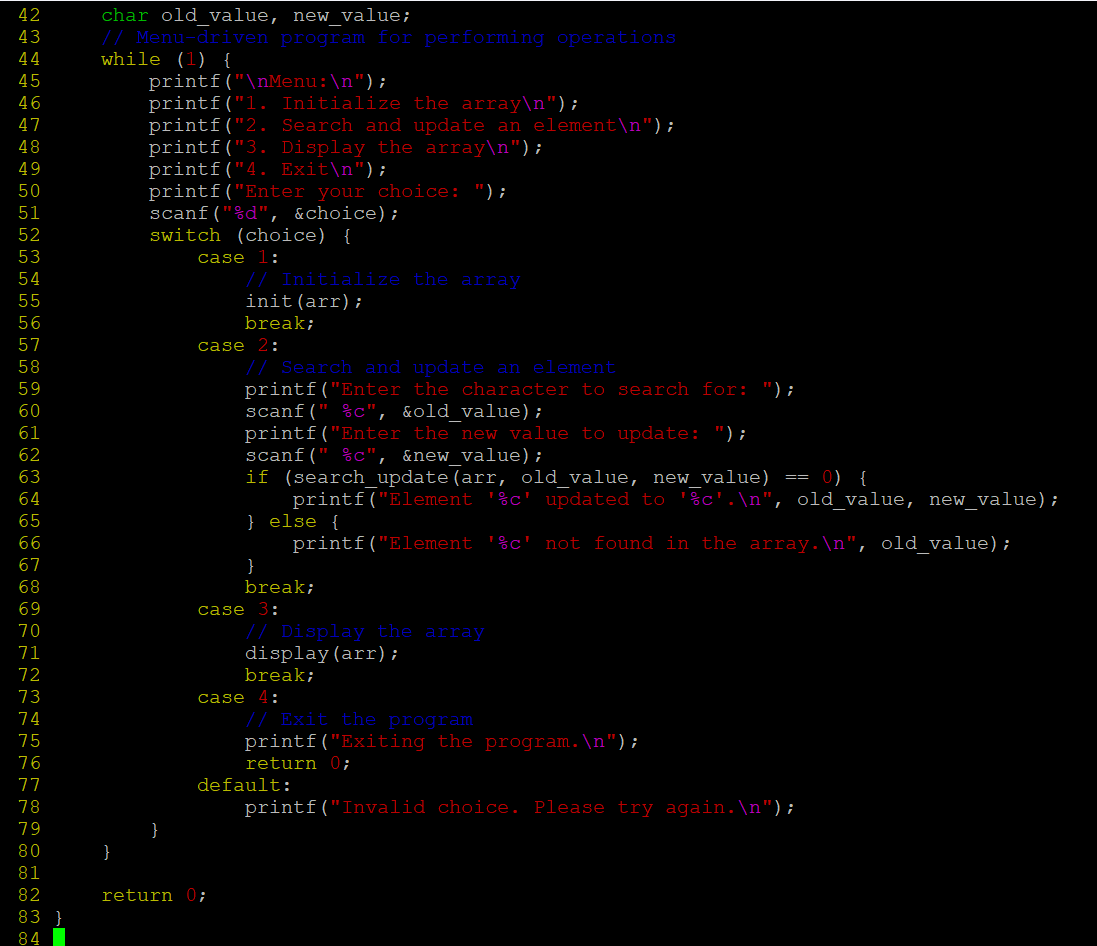
**a. init() - initialize the array and return 0**

**b. search\_update() – search for a given element in array and if found update it to given value and return 0 else return 1**

**c. display() – traverse and display array contents**

**For the functions, pass array and other required arguments to functions and return as per requirem**





A screenshot of a computer program

Description automatically generated

