is this copy work on the array of int or not

Yes, the concept of copying values upon initialization works similarly for arrays of other data types, including `int`.

Let's look at an example with an array of integers:

1. Initialization of an Integer Array:

When you write:

```
int arr[] = {1, 2, 3, 4, 5};
```

The compiler does the following:

- Allocates memory for the array `arr` to hold five integers.
- Initializes the allocated memory with the provided values.

Visualizing it:

```
Memory Address:    0x400    0x404    0x408    0x40C    0x410
Value:                 1          2          3          4          5
Name:                arr (starts here)
```

(Note: I'm assuming an `int` is 4 bytes, hence the address increments of 4. The exact size of an `int` can vary based on the platform and compiler.)

2. Pointer to an Integer:

If you were to declare a pointer and assign it the address of the array:

```
int *p = arr;
```

`p` would store the address of the first element of `arr`:

```
Memory Address:    0x500
Value:             0x400
Name:                 p
```

3. Copying Array of Integers:

Just as with the string example, if you create a new integer array and want to copy values from `arr`, you'd do something like:

```
int copy[5];
for (int i = 0; i < 5; i++) {
    copy[i] = arr[i];
}
```

This would create a new area in memory for the `copy` array and then transfer the values from `arr` into `copy`.

In Summary:

The principle of copying values upon array initialization or assignment holds true regardless of the data type of the array. Whether you're working with an array of chars, ints, floats, or any other type, the behavior is consistent: memory is allocated for the array, and values are stored in that memory. If you want to copy values from one array to another, you'd need to do it element by element (unless you use functions or methods that perform such copying for you).

more info to understand

Certainly! Let's go deeper into arrays of integers, and explore how they work in memory.

Arrays of Integers: A Closer Look

1. **What is an Array?**

   An array is a contiguous block of memory that holds elements of a specific type. If you declare:

   ```
   int arr[3];
   ```

   You are asking the compiler to reserve memory for 3 integers, placed one after the other in memory.

2. **Initialization**

   When you initialize the array at the time of declaration:

   ```
   int arr[3] = {10, 20, 30};
   ```

   The memory layout looks something like this (considering `int` as 4 bytes for simplicity):

   ```
   Memory Address:    0x1000   0x1004   0x1008
   Value:                  10        20        30
   ```

3. **Memory Allocation**

   Arrays are allocated in contiguous memory blocks. This contiguous allocation is why you can use pointers to navigate and access array elements. For example:

   ```
   int *p = arr;
   printf("%d", *p);        // Prints 10
   printf("%d", *(p + 1));  // Prints 20
   ```

   Here, the pointer `p` holds the memory address `0x1000`. When you increment the pointer (`p + 1`), it moves by the size of an integer (4 bytes in this case) to address `0x1004`.

4. **Copying Arrays**

   Arrays do not support direct assignment in C. This means if you try to do:

   ```
   int arr1[3] = {10, 20, 30};
   int arr2[3];
   arr2 = arr1;  // Error! Direct assignment not supported.
   ```

The above code will raise a compilation error. To copy an array, you need to manually copy each element, typically using a loop:

```
for (int i = 0; i < 3; i++) {
    arr2[i] = arr1[i];
}
```

5. **Comparing Arrays**

   Just as direct assignment is not supported, direct comparison (`arr1 == arr2`) is also not supported for arrays. You'll need to compare element by element.

6. **Arrays and Pointers**

   While arrays and pointers are different concepts, arrays are closely related to pointers. The name of an array is essentially a constant pointer to the first element. However, there are key differences:

   7. **Size**: Using `sizeof(arr)` gives the total size of the array (e.g., for `int arr[3]`, it will return 12 on a system where `int` is 4 bytes). Using `sizeof(p)`, where `p` is a pointer, will give the size of the pointer (often 4 or 8 bytes depending on the architecture).

   8. **Memory**: The memory for the array is allocated either on the stack (for local variables) or in the data segment (for global/static arrays). Pointer variables just hold an address and can point anywhere, including the stack, heap, or other memory regions.

   9. **Modifiability**: You can modify where a pointer points, but you can't change the memory location of an array.

Wrap-Up

Understanding the nuances of arrays in C requires distinguishing between the array itself (a block of memory) and the ways we can access and manipulate that memory (using array indices or pointers). The key idea is that the array provides a fixed, contiguous block of memory, and how we interact with that block can vary based on our needs and the constructs the C language provides.

> Modifiability: You can modify where a pointer points, but you can't change the memory location of an array.

Right, let's expand on that point:

Modifiability of Pointers vs Arrays:

Pointers:

When you declare a pointer, you're essentially declaring a variable that's designed to store a memory address.

For instance:

```
int *p;
int x = 10;
int y = 20;
```

Here, `p` is uninitialized and doesn't point to a meaningful location yet. But you can make it point to `x`:

```
p = &x;  // p now holds the address of x
```

And later, if you want, you can change `p` to point to `y`:

```
p = &y;  // p now holds the address of y
```

This shows the **modifiability** of a pointer. You can change the memory address that the pointer holds (or points to) anytime you want.

Arrays:

When you declare an array:

```
int arr[5];
```

You're asking the compiler to allocate a block of memory to hold 5 integers. The name `arr` represents the address of the first element of this block.

You can use this name to access the array elements:

```
arr[0] = 10;  // Set the first element to 10
```

But what you cannot do is make `arr` point somewhere else. This line:

```
arr = some_other_address;  // This is invalid and will cause a compiler error
```

Is not allowed. The reason is that `arr` is not just a variable holding an address (like a pointer); it's a label representing a specific location in memory where the array starts.

In Essence:

- **Pointers** are like movable arrows; they can be redirected to point to different memory locations.
- **Arrays**, on the other hand, are like fixed signs. Once they are planted in a location, they stay there, representing that specific block of memory. You can change the content (values inside the array), but you can't move the entire array to another memory location or make the array name point somewhere else.

From ChatGPT, designed with FancyGPT