Arrays

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Prelude

- We are now entering the next phase of the course
- It requires having a solid understanding of earlier concepts
- Ensure you know how to design loops and how to trace codes

Scenario /1

- Write a program that asks the user to enter **2 whole numbers**
- Print the numbers in **reverse order** in which they were entered
- How about 10 numbers?
- How about 1,000 numbers?

Scenario /2

- How about 1,000,000 numbers?

- You get the point?

Sample Run

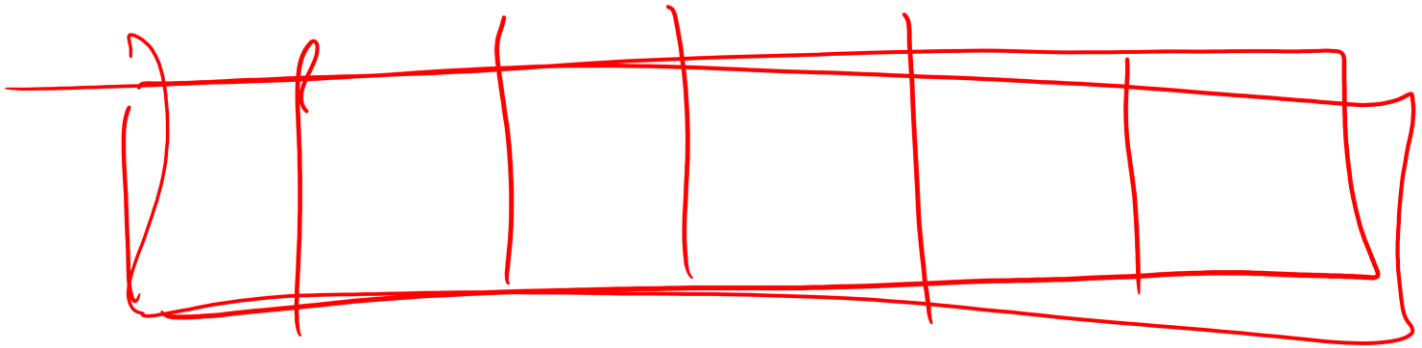
```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
```

Data Structures

- A way to **group** and **organize** related data items together

- Provides efficient ways for **storing**, **accessing** and **managing** data

Arrays



- Collection of fixed-size data items with **same data type** and **name**
- Think of it as a “list” with a **predefined** number of slots
- Each slot is referred to as an **element** accessed by an **index**

Declaring an Array in C

Similar to declaring a typical variable

You specify the data type, the name, and the **number of elements** (i.e., the number of slots you want to reserve)

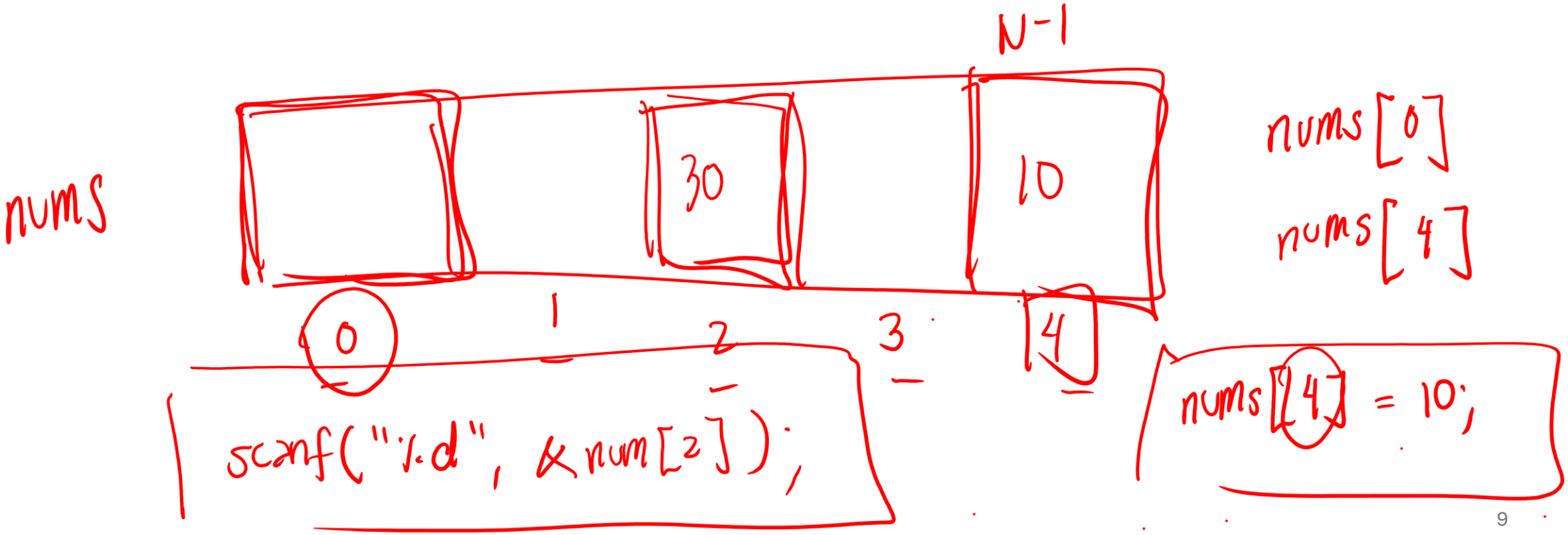
Syntax:

```
int nums[5];
```

Discussion

int nums[5];

If all elements **share the same name**, how do we refer to (or access) an individual element?



Accessing an Individual Element /1

- Provide a special number that refers to its “position” in the list
- Formally referred to as **subscript**
- Think of it as an ID number, some refer to it as *index*
- Specify using `[ ]` after the array’s name

Accessing an Individual Element /2

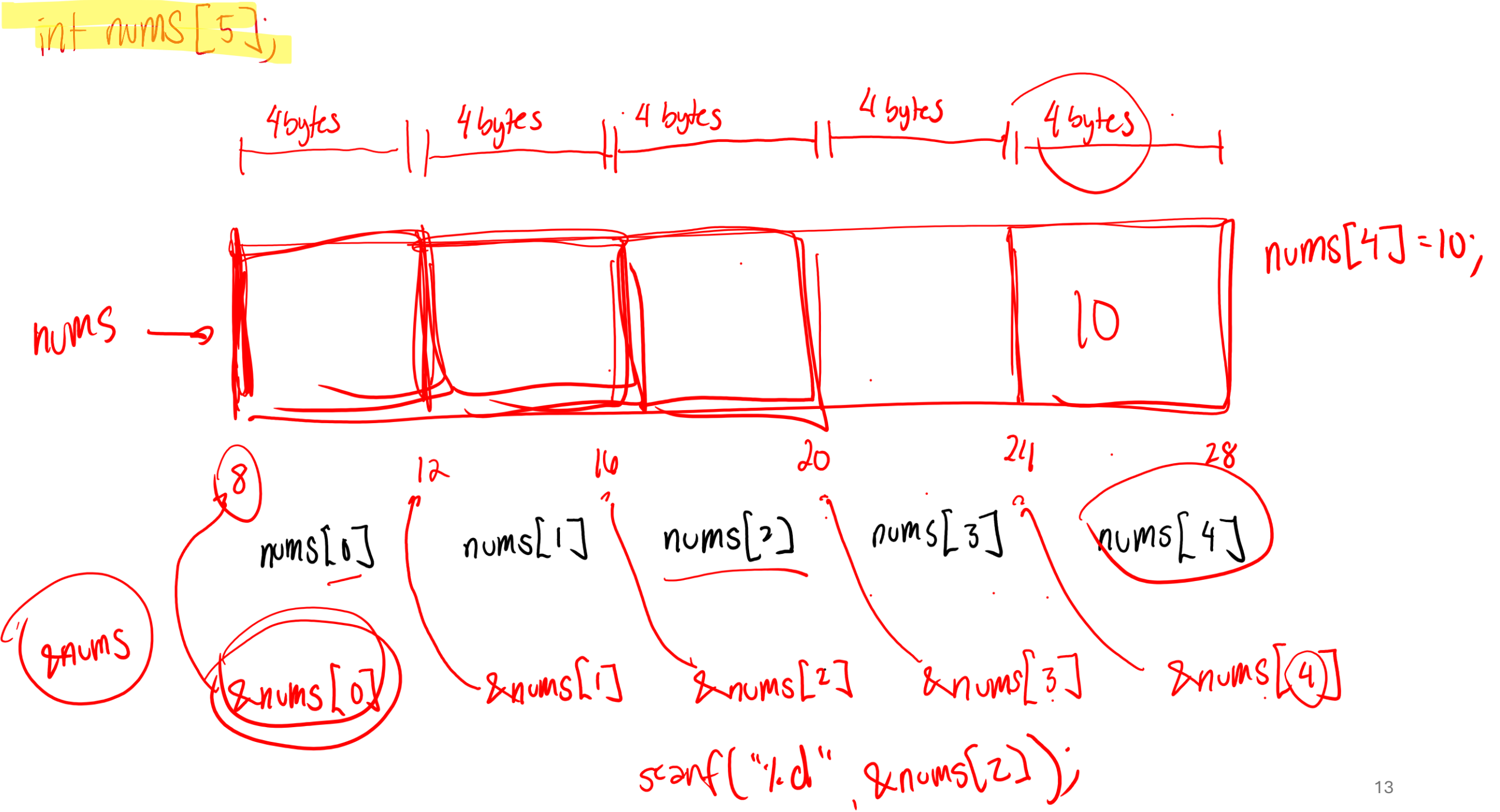
- For now, just remember that the first ID number is **0**

- Up until **N-1** (where **N** is the number of slots; size of the array)

Practice

Declare an array of integers with size 5. Afterward, print the **address of** each element of the array.

int nums[5];


4 bytes
4 bytes
4 bytes
4 bytes
4 bytes
nums
10
nums[4] = 10;
8
12
16
20
24
28
nums[0]
nums[1]
nums[2]
nums[3]
nums[4]
&nums
&nums[0]
&nums[1]
&nums[2]
&nums[3]
&nums[4]
scanf("%d", &nums[2]);

Notes

- It looks like the addresses are “next to each other”
- Notice the difference **between the addresses of two elements**?
- Two or more **adjacent** memory cells (i.e., contiguous)

Discussion

What about the **address of** the array itself?

Notes /1

- Recall that a variable name is an alias to a memory location

- The array elements are stored contiguously in memory, starting at the address of the first element

Notes /2

- The index of the first element is 0 because its **offset** from the base address is 0

- Therefore, we can view index as simply a ***displacement*** from the starting address

Your Turn!

What if we have an array with different data type?

- Say `double`?
- How about `float`?
- How about `char`?

```c
#include <stdio.h>

int main() {
    double nums[10];
    int i;

    printf("Size Of Double: %d\n", sizeof(double));
    for(i = 0; i < 10; i++) {
        printf("Address of nums[%d]: %p %lu\n", i, &nums[i],
            (unsigned long)&nums[i]);
    }

    printf("Address of Array: %p %lu\n", &nums, (unsigned long
        )&nums);

    return 0;
}
```

```
Size Of Double: 8
Address of nums[0]: 0x7ffc4d8c8900 140721609541888
Address of nums[1]: 0x7ffc4d8c8908 140721609541896
Address of nums[2]: 0x7ffc4d8c8910 140721609541904
Address of nums[3]: 0x7ffc4d8c8918 140721609541912
Address of nums[4]: 0x7ffc4d8c8920 140721609541920
Address of nums[5]: 0x7ffc4d8c8928 140721609541928
Address of nums[6]: 0x7ffc4d8c8930 140721609541936
Address of nums[7]: 0x7ffc4d8c8938 140721609541944
Address of nums[8]: 0x7ffc4d8c8940 140721609541952
Address of nums[9]: 0x7ffc4d8c8948 140721609541960
Address of Array: 0x7ffc4d8c8900 140721609541888


=== Code Execution Successful ===
```

```c
#include <stdio.h>

int main() {
    char letters[10];
    int i;

    printf("Size Of Char: %d\n", sizeof(char));
    for(i = 0; i < 10; i++) {
        printf("Address of letters[%d]: %p %lu\n", i,
            &letters[i], (unsigned long)&letters[i]);
    }

    printf("Address of Array: %p %lu\n", &letters, (unsigned
        long)&letters);

    return 0;
}
```

```
Size Of Char: 1
Address of letters[0]: 0x7ffd981e0202 140727155556866
Address of letters[1]: 0x7ffd981e0203 140727155556867
Address of letters[2]: 0x7ffd981e0204 140727155556868
Address of letters[3]: 0x7ffd981e0205 140727155556869
Address of letters[4]: 0x7ffd981e0206 140727155556870
Address of letters[5]: 0x7ffd981e0207 140727155556871
Address of letters[6]: 0x7ffd981e0208 140727155556872
Address of letters[7]: 0x7ffd981e0209 140727155556873
Address of letters[8]: 0x7ffd981e020a 140727155556874
Address of letters[9]: 0x7ffd981e020b 140727155556875
Address of Array: 0x7ffd981e0202 140727155556866


=== Code Execution Successful ===
```

Revisiting: Scenario

- Let’s solve the problem where we print the list of 5 numbers in reverse order

- How do we solve this?

Sample Run

```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
```

10
20
30
40
50

Planning the Solution /1

- Can we divide the entire problem into sections or parts?
- If the goal is to print the numbers in reverse, we need to know those numbers
- Therefore, we must ask the user to enter the numbers first

Planning the Solution /2

Objective 1:

Obtain the numbers from the user and store them

Objective 2:

Print the numbers in reverse order

Planning the Solution /3 – Objective 1

- If applicable, try to scale down the problem
- I know that I am going to repeatedly perform the same task…
- Asking for user input and storing it
- So, there will be a loop!

Planning the Solution /4 – Objective 2

- Now, we need to find a way to access those numbers again!

- However, this time, we want to do it in reverse order

Discussion /1

- Notice how the size of the array is all over the code?
- That is a **bad programming practice**
- Why?
- Say, I changed my mind… from 5 to just 100 elements?

Discussion /2

- The value you saw is also known as a **magic number**

- They *magically* appeared in our code without context

Recall: Macro Constants

A preprocessor directive using **`#define`**

Syntax:

It **does not** have a semicolon

Think of it as a **find and replace**

Common Array Operations

- Iterating through each element in the array
- You are repeatedly performing the same task…
- Accessing a single element
- So, there will be a loop!

Practice

Extend the previous program such that it also prints the **sum** of all the numbers entered by the user.

Sample Run

```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
Sum: 150
```

sum = 0;

Practice

Extend the previous program such that it also prints the **average** of all the numbers entered by the user.

Sample Run

```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
Sum: 150
Average: 30.00
```

Practice

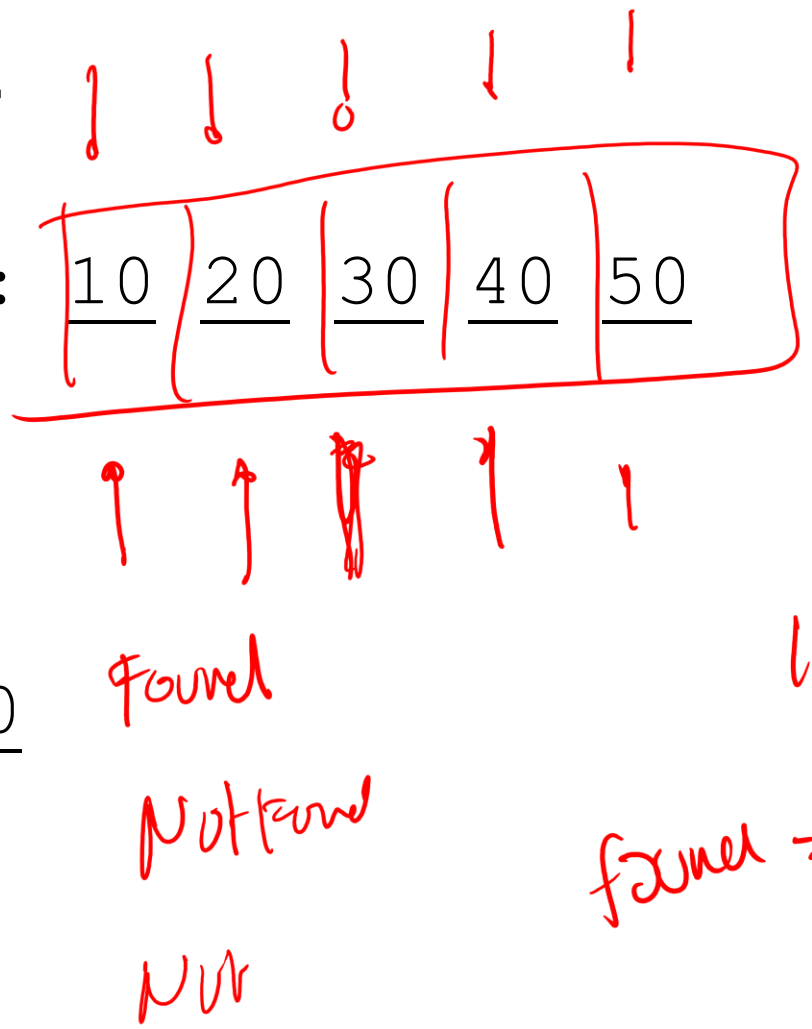
Extend the previous program such that after entering all the numbers, the user is prompted to enter a whole number called **`query`**. The program prints “Found” if **`query`** is in the array. Otherwise, it prints “Not Found”.

Sample Run /1

```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
Sum: 150
Average: 30.00
Enter Query: 30
Found
```

Sample Run /2

```
Enter 5 Numbers:
50 40 30 20 10
Sum: 150
Average: 30.00
Enter Query: 100
Not Found
```



Discussion

- The formal name of the search that we performed is **linear search**
- In our approach, we utilized a **flag variable** to indicate whether we found **`query`**
- We also utilized **`break`** to get out of the loop

Your Turn!

Extend the program so that it prints both the **min** and **max** values.
You may assume that the user will enter distinct values.

Sample Run

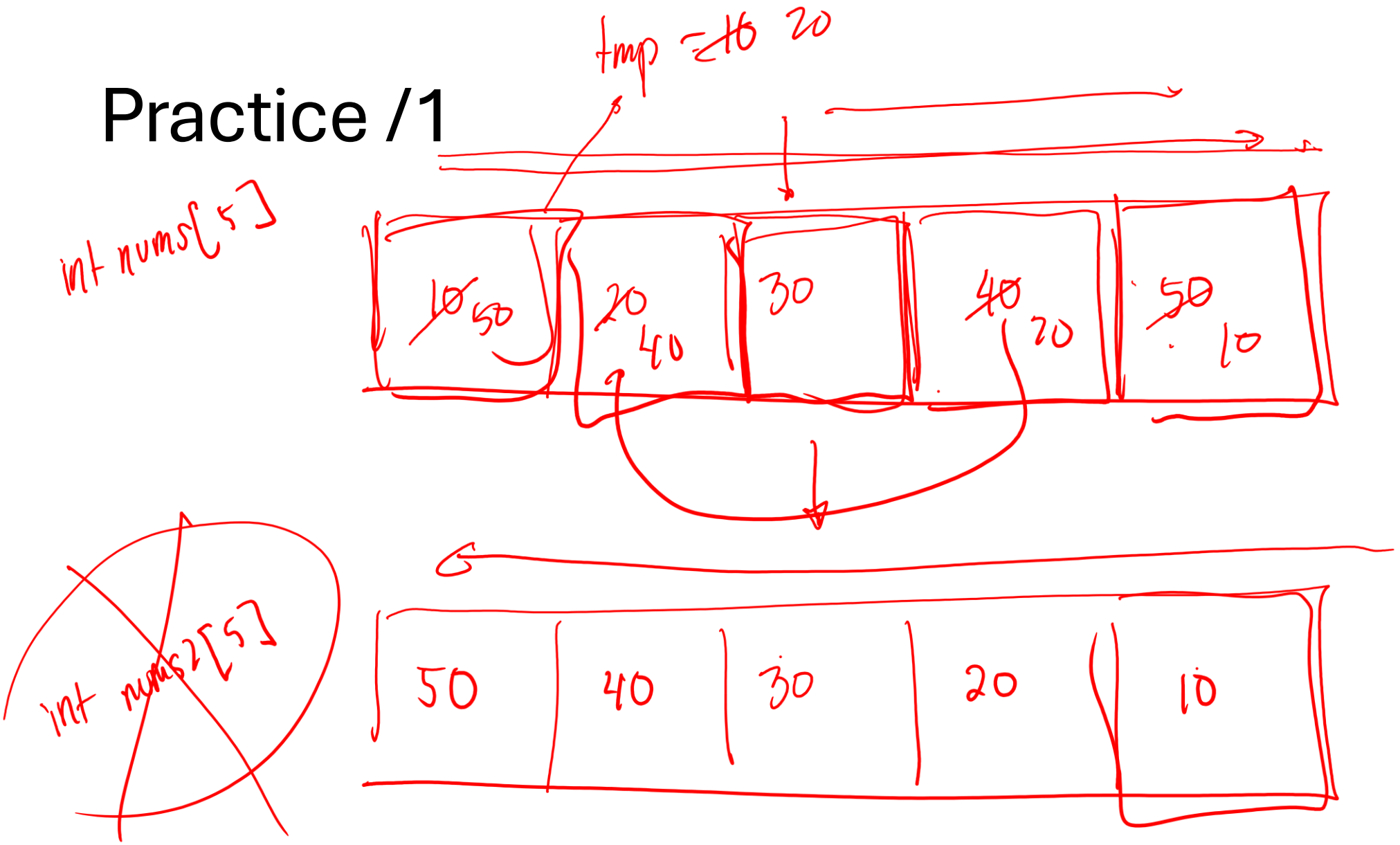
```
Enter 5 Numbers: 10 20 30 40 50
50 40 30 20 10
Sum: 150
Average: 30.00
Enter Query: 100
Not Found
Min: 10
Max: 50
```

Discussion

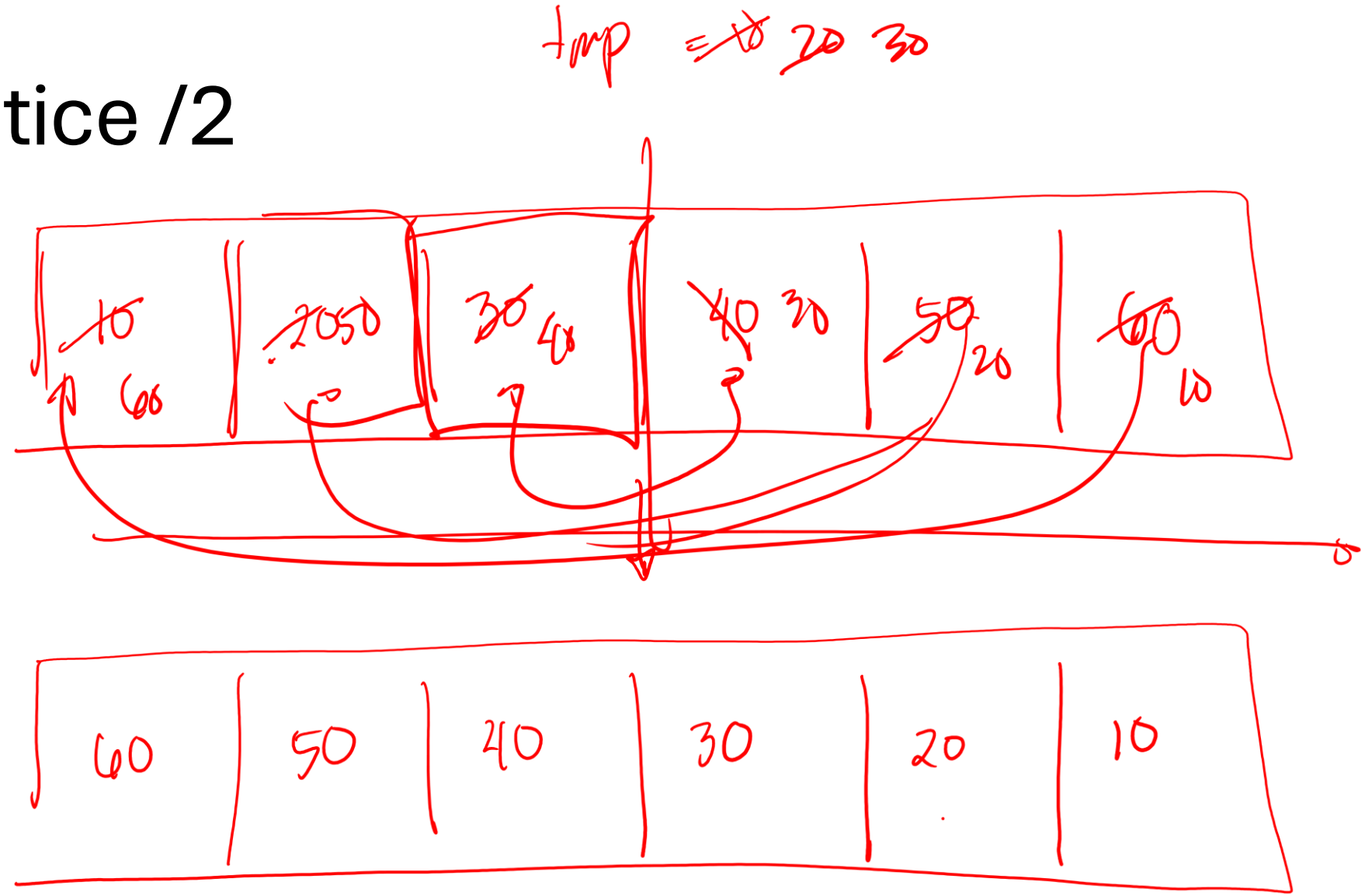
- Our approach to printing the values in reverse order was to traverse the array backward

- But what if we want to rearrange the array itself in reverse order?

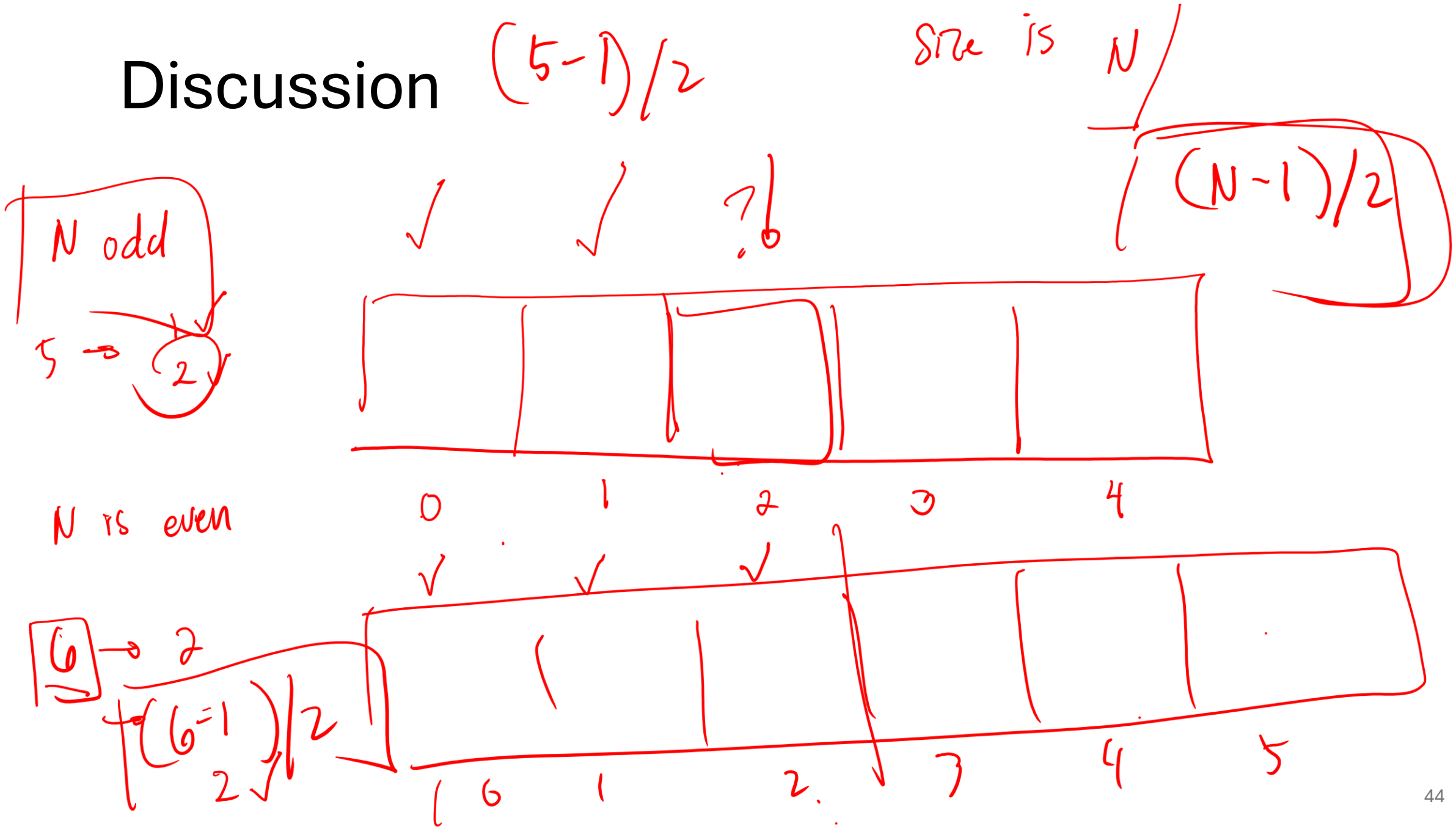
Practice /1


int nums[5]
tmp = 10 20
10 50
20 40
30
40 20
50 10
int nums2[5]
50
40
30
20
10

Practice /2



Discussion



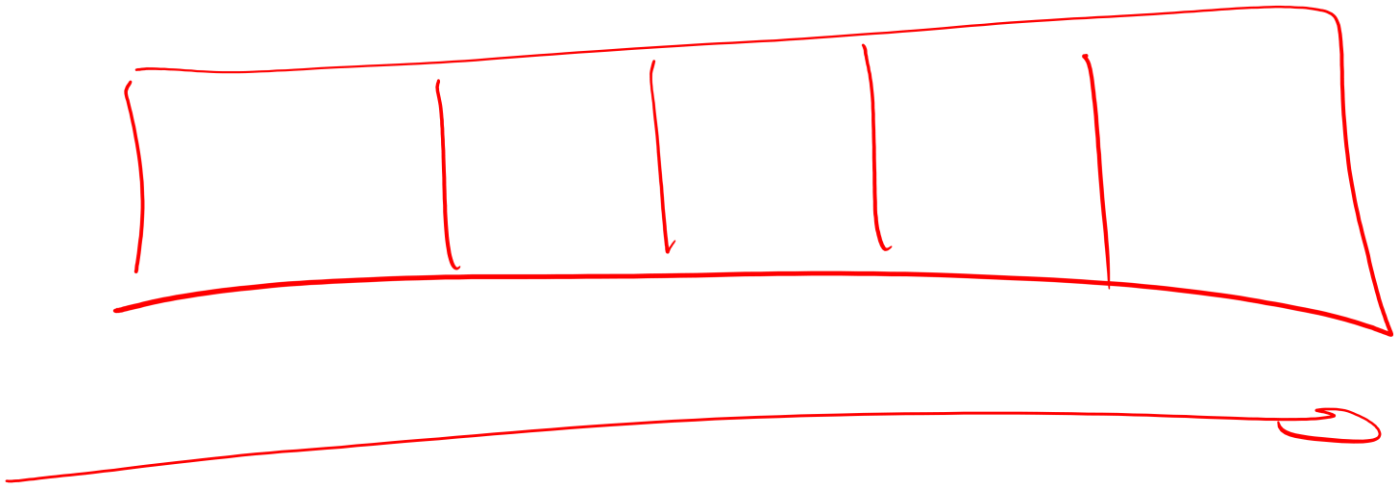
Notes

- Notice the existence of the *swapping behavior* in our solution?
- Again, this will involve the use of a **temporary variable**
- Additionally, notice the *behavior* of the loop

Discussion

What do you think are the initial values of each element after declaring an array?

```
int nums[5];
```



Initializing an Array /1

Like an ordinary variable, you can set **initial values** of an array

We provide an **initializer list**

The syntax is:

```
int nums[5] = {10, 20, 30, 40, 50};
int arr [] = {10, 20, 30, 40, 50};
```

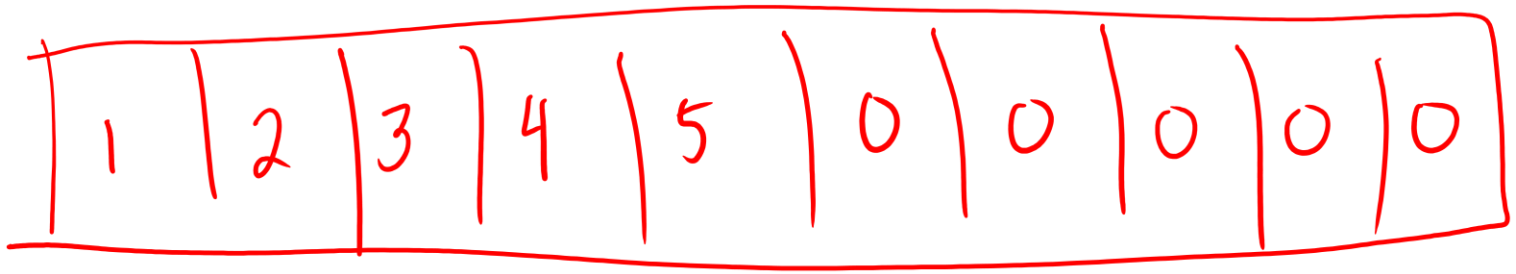
the size of the array is inferred from the list

Initializing an Array /2

Why did it work even if we did not provide the size?

What if we provided both the size and the initial values?

```
int nums[10] = {1, 2, 3, 4, 5};
```



Initializing an Array /3

If the size is defined and the initializer list provided has a smaller size, the **default value** of the data type will be used

Default Values

`int` 0

`double` 0.0

`float` 0.0

`char` '\0' 0

null character

Practice

What are the elements of the following array?

```
int nums[10] = {0};
```

0, 0, 0, 0, 0, 0, 0, 0, 0, 0

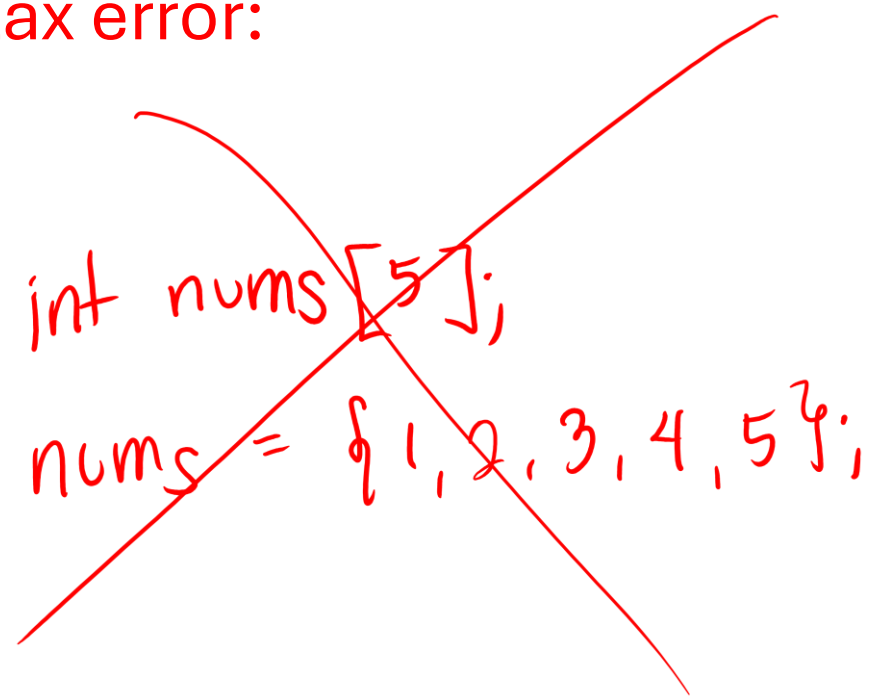
```
int arr[10] = {1};
```

1, 0, 0, 0.

Initializing an Array /4

Important to note that this only works when array is declared

This is a syntax error:



Discussion

- Do you see how our **`main()`** function is breaking the principle of modularity we previously discussed?

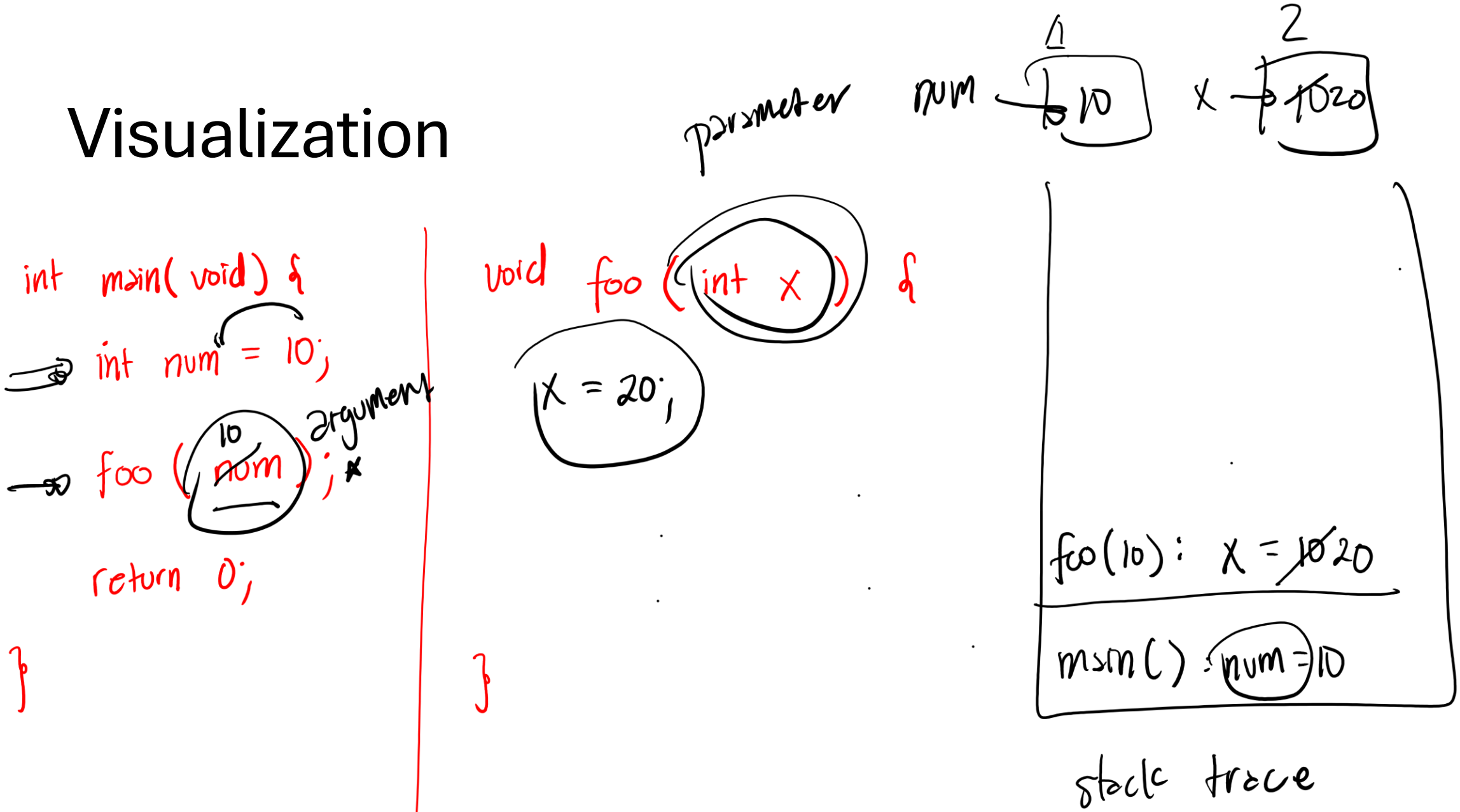
- Can we reorganize our code?

Recall: Functions

Argument is the actual value that we are passing

Parameter is the variable receiving the value being passed

Visualization



Functions with Array Arguments

Provide the function prototypes of the following functions that:

- ✓ Computes the sum of all the elements

```
int compute_sum(int arr[], int size);
```

- ✓ Computes the average of all the elements

```
double compute_average(int arr[], int size);
```

- ✓ Searches for a query value

```
int search_value(int arr[], int size, int query);
```

Notes

- When passing an array to a function, we pass only the array name without brackets

- Passing an array element to a function is different (e.g., **`arr[i]`**)

- In addition to passing the array, we also pass the size of the array as an additional argument to the function

Practice

Write a function named **`print_array()`** that takes two parameters: an array of integers and an integer representing the array's size. It should print all the elements on a single line followed by a newline.

Discussion

What do you think the array name represents when printed using **`printf`**?

Notes

In most expressions, the name of the array ***decays*** to the address of its first element

Practice

Write a function named **`square()`** that takes two parameters: an array of integers and an integer representing the array's size. The function should update the array so that each element is replaced with its square.

Sample Run

```
Enter 5 Numbers: 10 20 30 40 50
100 400 900 1600 2500
```

Notes /1

- When an array is passed as an argument to a function, what is passed is an **address**

- The function operates on the original array, not a separate copy

Notes /2

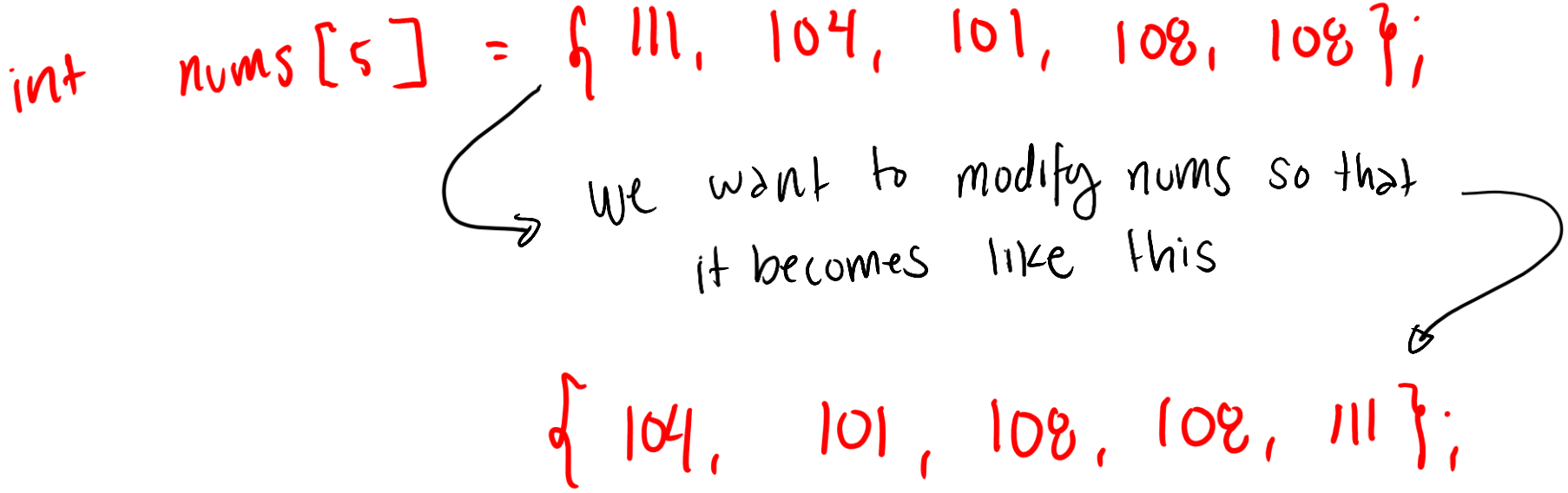
- Any changes made inside the function will be visible to the caller
- You can think of this as the array being shared between the calling and called functions
- Sometimes, this behavior is described as ***pass by reference****

Your Turn!

Write a function named **`reverse_array()`** that takes two parameters: an array of integers and an integer representing its size. The function should reverse the order of the array's elements.

Challenge

Assume that you have an array of 5 whole numbers. Write a program that shifts all the elements of the array to the left by **1**. In this case, the first element becomes the new last element.



Discussion

- This operation is known as **circular shift**

- It is possible to shift by **`k`** positions to the left

Challenge

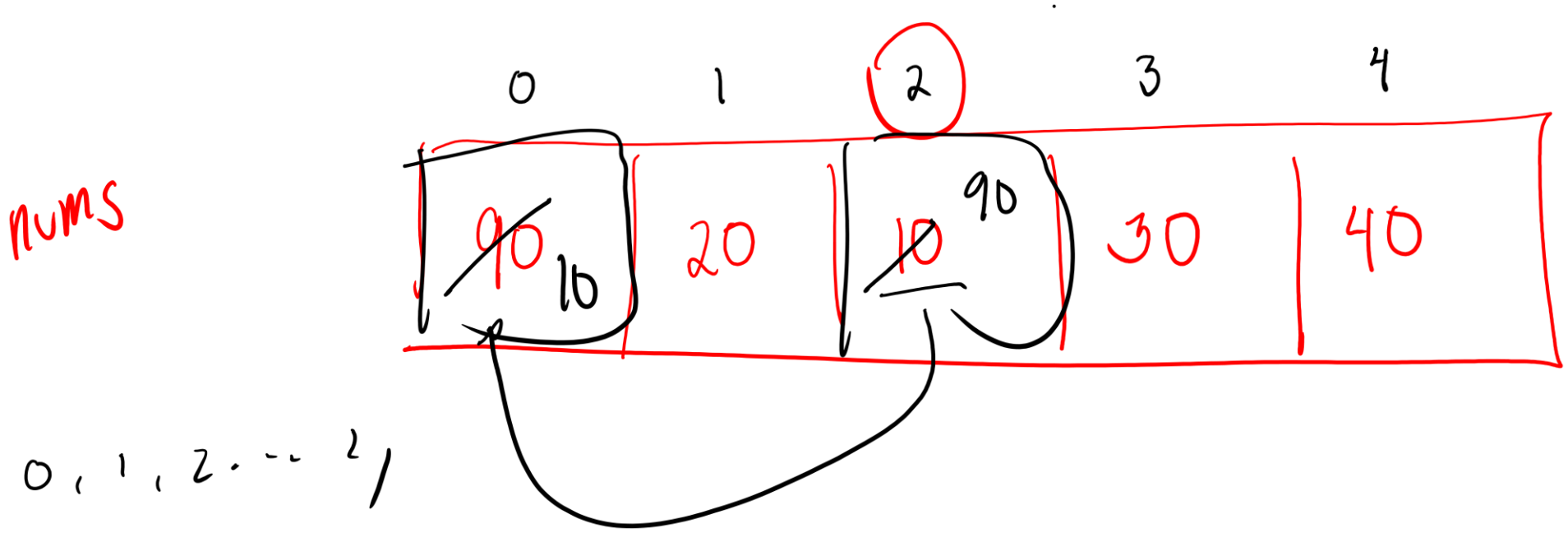
Implement the circular shift by **`k`** positions. Assume that you have an array of 5 integers. The user provides the value for **`k`**. Afterward, print the values of the array after **`k`** left shifts.

Discussion /1

nums [2]

tmp = 90

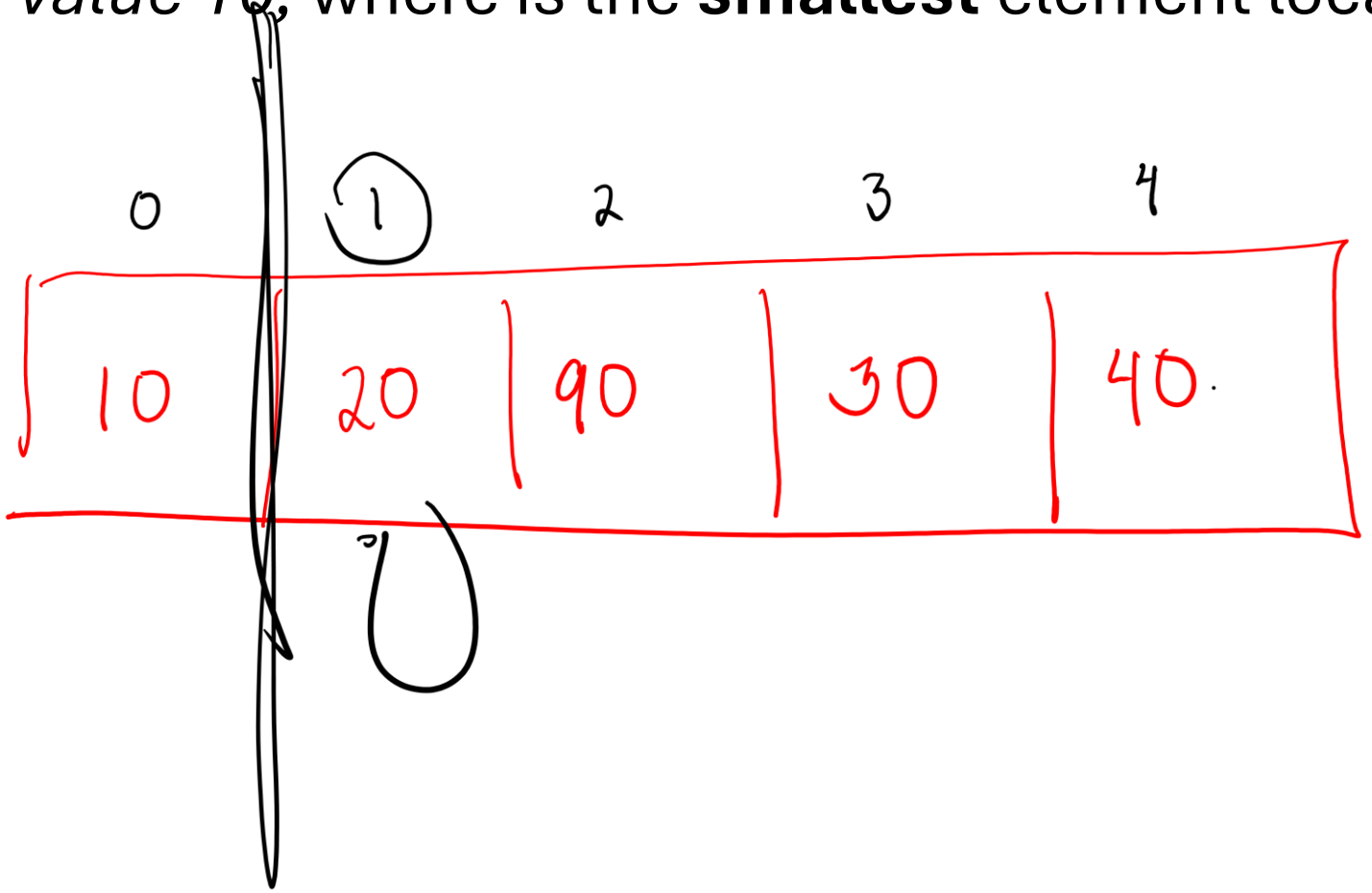
Where is the **smallest** element located? What is its **value**?



Discussion /2

tmp = 20

Ignoring the value 10, where is the **smallest** element located?

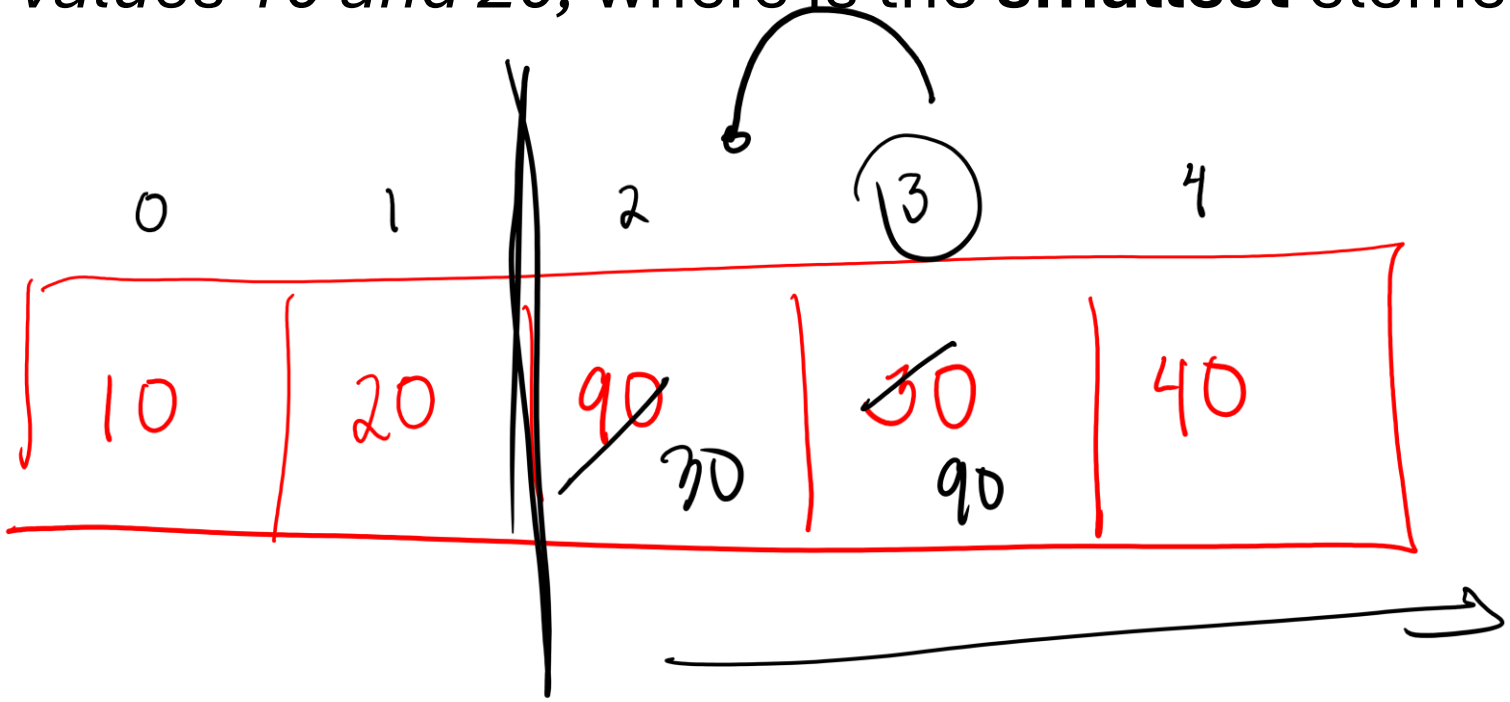


1, 2, 3, 4

Discussion /3

tmp = 90

Ignoring the values 10 and 20, where is the **smallest** element located?

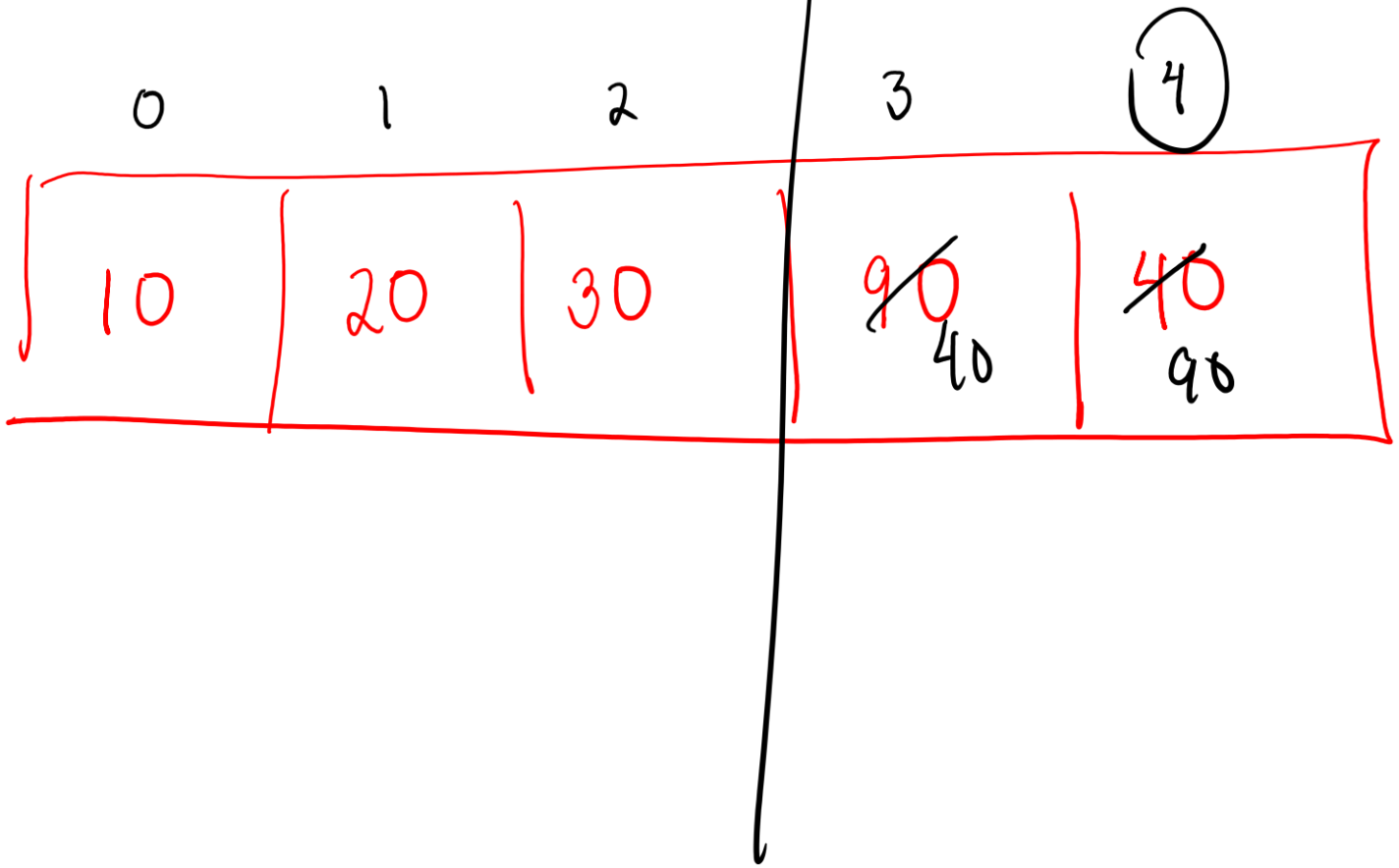


2, 3, 4

Discussion /4

tmp = 90

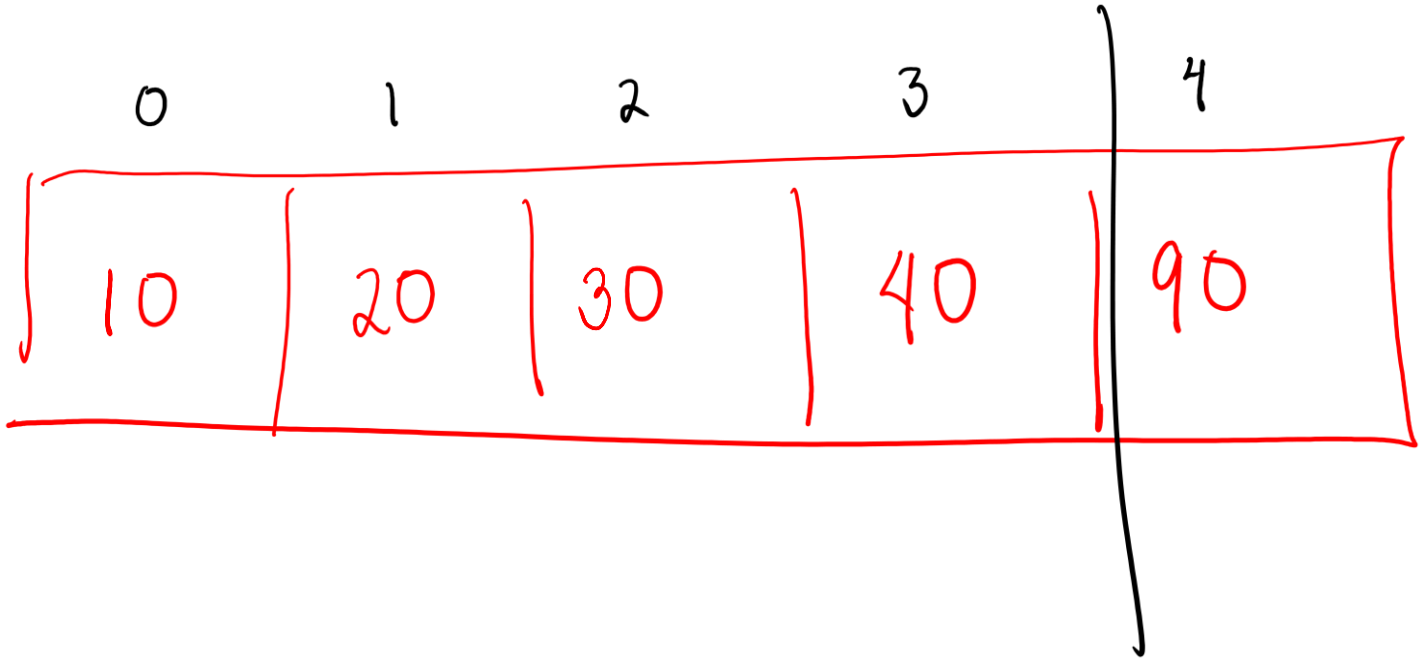
Ignoring the values 10, 20, and 30, where is the **smallest** element located?



3, 4

Discussion /5

Ignoring the values 10, 20, 30 and 40, where is the **smallest** element located?

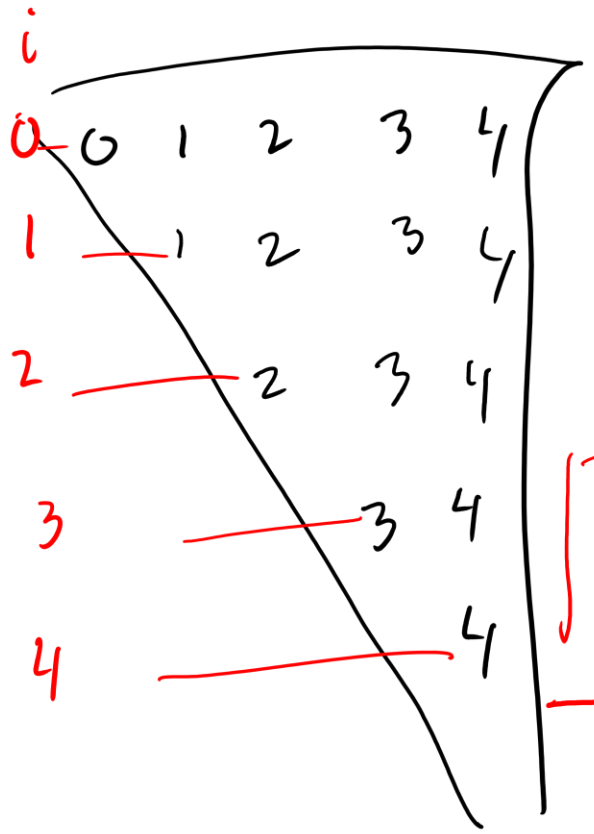


4

Discussion /6

for loop → # of rounds ✓ N

for loop → find the min




```
for ( int i=0 ; i <N ; i++ )
    for ( int j =i ; j <N ; j++ )
```

0	1	2	3	4
10	20	30	40	90

Notes /1

- Our approach was to keep on searching for the smallest element
- Once we find it, we swap it with the leftmost element of the part we are currently working on
- Each time we do this, we can simply ignore the elements already placed earlier (i.e., swapped)

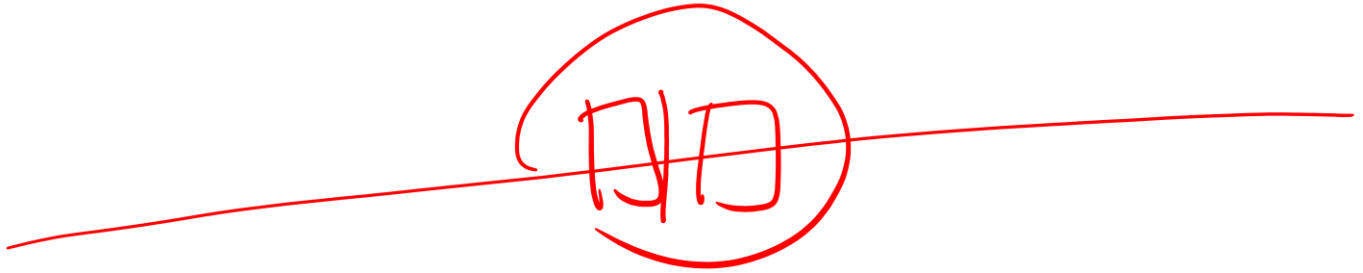
Notes /2

- If we keep repeating the process, we will start to notice a gradual change in how the array looks

- We just witnessed an example of an elementary sorting algorithm called **selection sort**

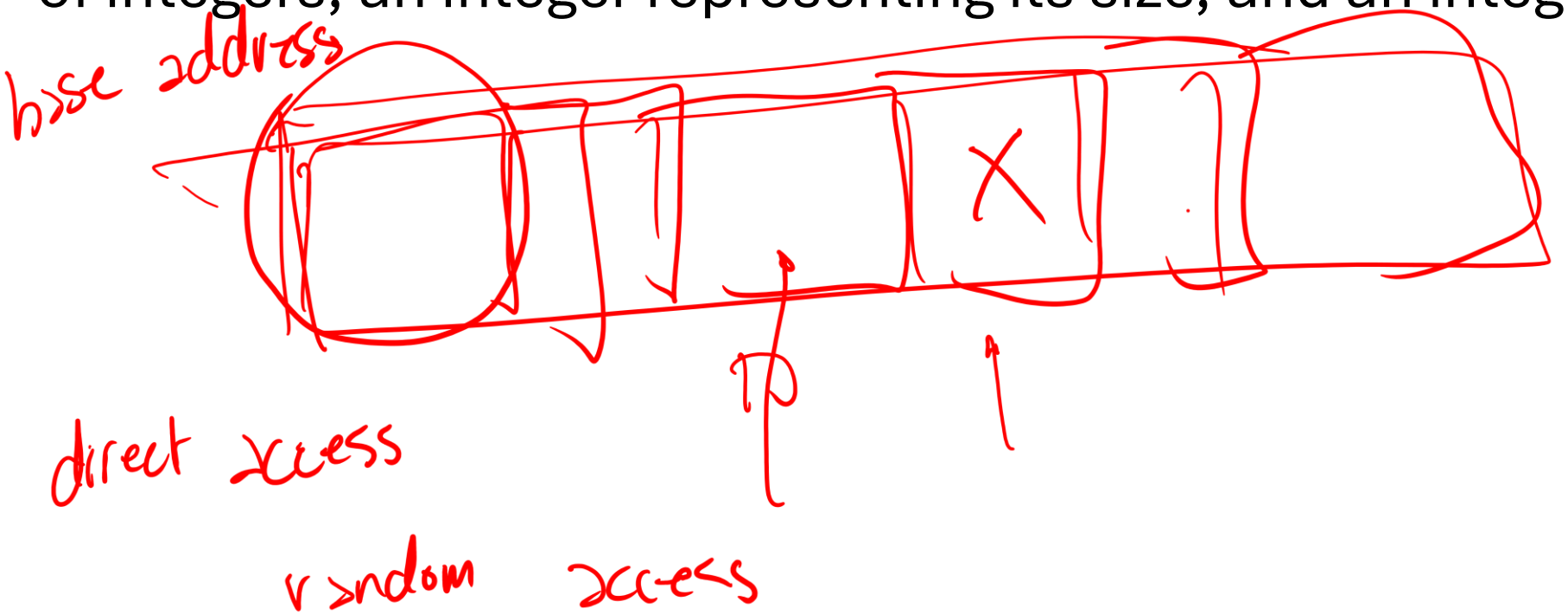
Challenge /1

Write a function named **`find_median()`** that returns a **`double`**. It takes two parameters: an array of integers and an integer representing its size. The function returns the median. If the size is even, simply return the average of the two middle elements.



Challenge /2

Write a function named **`find_kth_largest()`** that returns the **`k`**th largest element in an array. It takes three parameters: an array of integers, an integer representing its size, and an integer **`k`**.



Scenario

- Write a program that stores a list of whole numbers
- However, you **don’t know yet how many numbers** to store
- The user will be providing this at run time (i.e., it is a user input)
- How would you plan your solution?

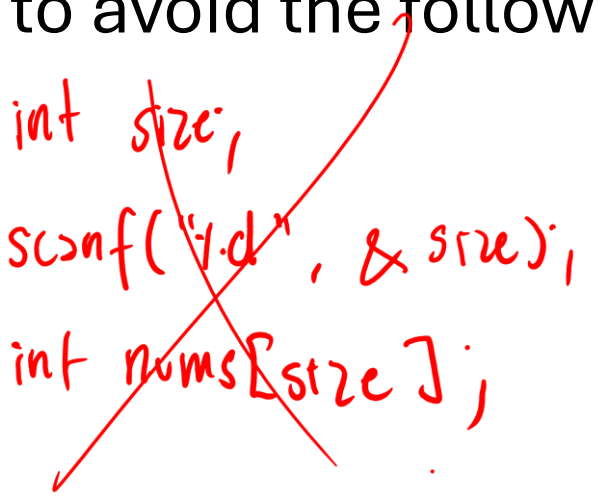
Fixed-Sized Arrays

int nums[10];

- Recall that for regular arrays, the compiler must know the array size at **compile time** (the size is fixed in the code)
- When the program runs, these arrays are stored in **stack memory** (automatic storage)
- An array's size is fixed and cannot be changed after it is created

Discussion /1

We want to avoid the following statement:



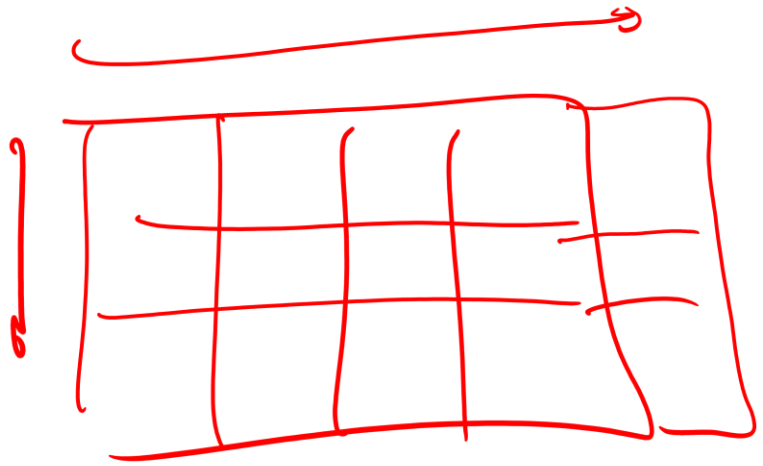
What happens if the user provides a very large number?

Discussion /2

The type of array we need is known as **variable-length array** as its size is determined at **runtime**

We will revisit this idea later when we learn about **dynamic memory allocation**

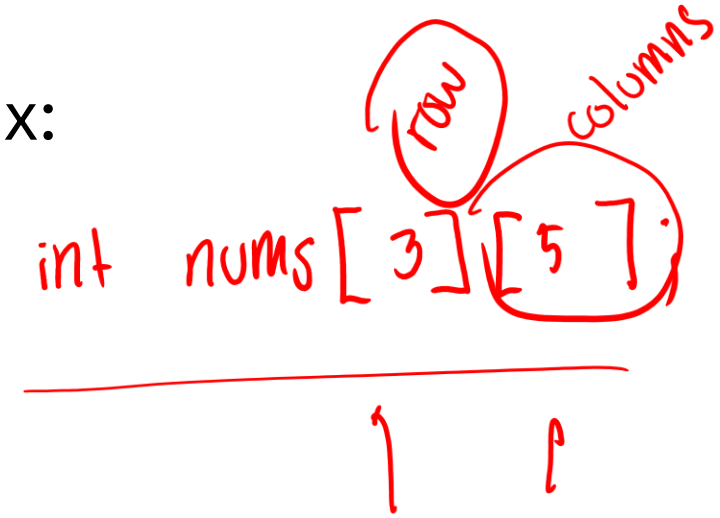
2D Arrays /1



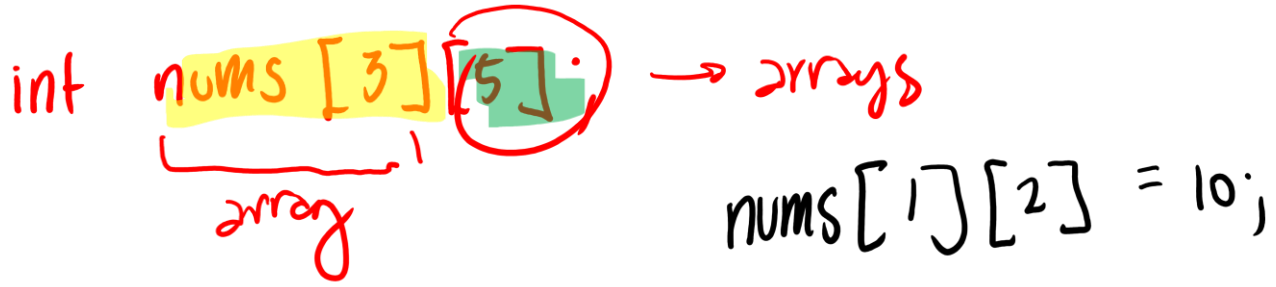
Arrays with two dimensions

Imagine them as a table (or **grid**) with rows and columns

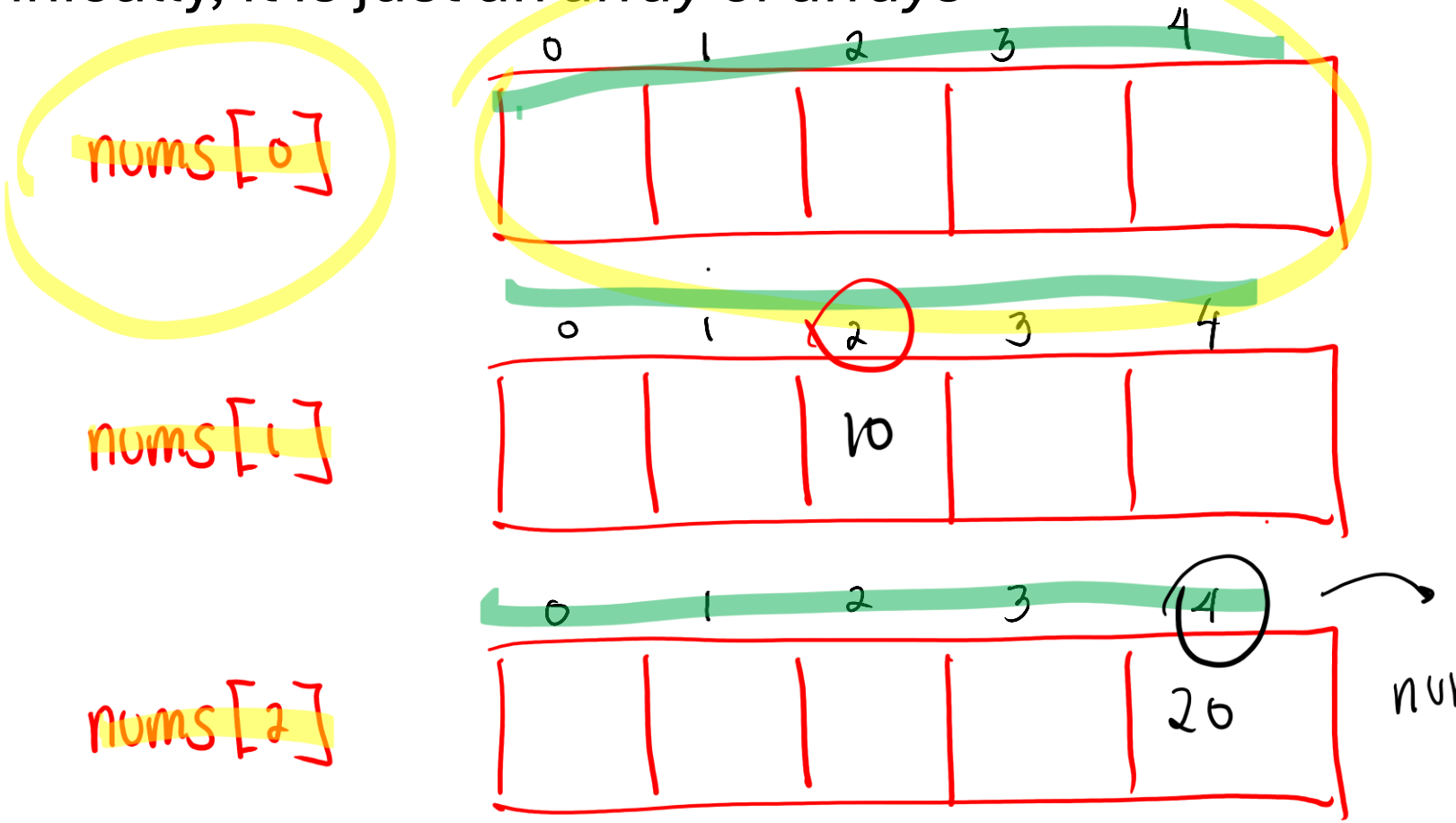
Syntax:



2D Arrays /2



Technically, it is just an *array of arrays*



Each element turns out to be an array

nums [2] [4] = 20;

2D Arrays /3

```
int nums[3][5];
```

	0	1	2	3	4
0					
1					
2					

Memory Layout of 2D Array

```
nums[0][0] = 6422156
nums[0][1] = 6422160
nums[0][2] = 6422164
nums[0][3] = 6422168
nums[0][4] = 6422172
```

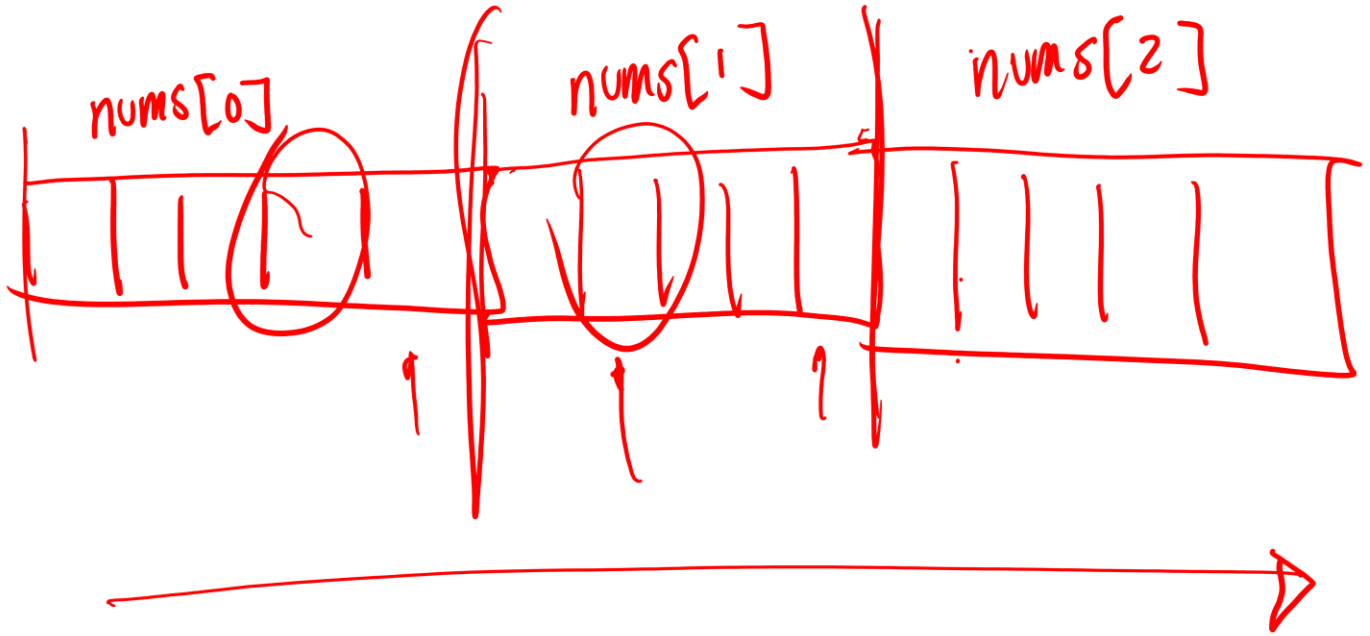
nums[0]

```
nums[1][0] = 6422176
nums[1][1] = 6422180
nums[1][2] = 6422184
nums[1][3] = 6422188
nums[1][4] = 6422192
```

nums[1]

```
nums[2][0] = 6422196
nums[2][1] = 6422200
nums[2][2] = 6422204
nums[2][3] = 6422208
nums[2][4] = 6422212
```

nums[2]



Initializing 2D Arrays /1

- Similar to a 1D array where we use an initializer list

- However, recall our visualization in the previous slide?

Initializing 2D Arrays /2

nums

10	20	30	90
90	70	10	20
40	50	20	50

```
int nums[3][4] = {
    { 10, 20, 30, 90 },
    { 90, 70, 10, 20 },
    { 40, 50, 20, 50 }
};
```

Traversing 2D Arrays /1

- Involves a **nested loop**

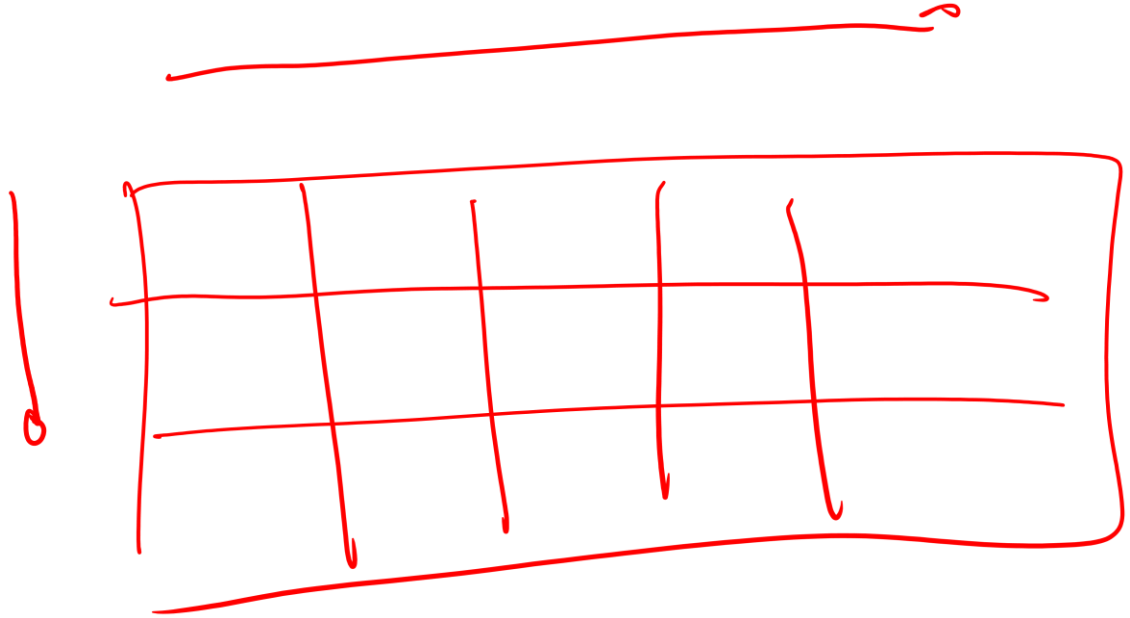
- Recall the following practice problem

Recall: Sample Run

```
Enter Two Numbers: 3 5
*****
*****
*****
```

Traversing 2D Arrays /2

- The **outer loop** is often associated with the **row**
- The **inner loop** is often associated with the **column**



Traversing 2D Arrays /2

- It is possible to swap the roles of the outer and inner loops
- Meaning, the **outer loop** can be for the **column**
- The **inner loop** can be for the **row**

0	10	20	30	90
1	90	70	10	20
2	40	50	20	50

```c
#include <stdio.h>
#define ROWS 3
#define COLS 4

int main(void) {
    int i;
    int j;
    int data[ROWS][COLS] = {
        {10, 20, 30, 90},
        {90, 70, 10, 20},
        {40, 50, 20, 50}
    };

    for(i = 0; i < COLS; i++) {
        for(j = 0; j < ROWS; j++) {
            printf("%d ", data[j][i]);
        }
        printf("\n");
    }

    return 0;
}
```

What is the output of the C program?

i	j	data[j][i]
0	0	data[0][0] => 10
0	1	data[1][0] => 90
0	2	data[2][0] => 40
1	0	20
1	1	70
1	2	50
2	0	
3	1	
3	2	
3	0	
3	1	
3	2	

Notes

When defining a function prototype that involves 2D arrays, the size of the second dimension needs to specified

More on this later when we discuss **pointers**

Challenge

Write a function named **`transpose()`** that accepts two parameters: a 2D square matrix and an integer representing its size. The function should transpose the matrix by swapping its elements across the main diagonal. You may assume that the matrix has a maximum capacity of 10 x 10.

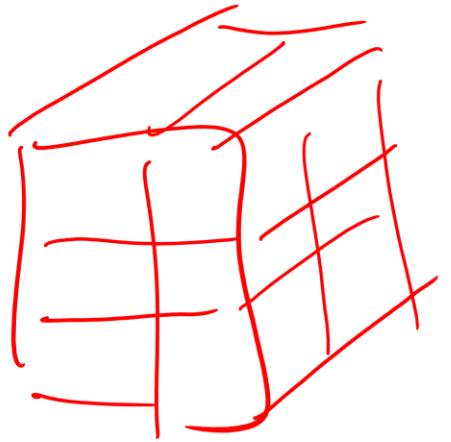
```
void transpose(int matrix[MAX_SIZE][MAX_SIZE], int size);
```

Sample Run

10	20	30
90	70	10
40	50	20

10	90	40
20	70	50
30	10	20

Multi-Dimensional Arrays /1



- You are not limited to 1D and 2D arrays
- You can have N-dimensional arrays
- The syntax for a 3D array:

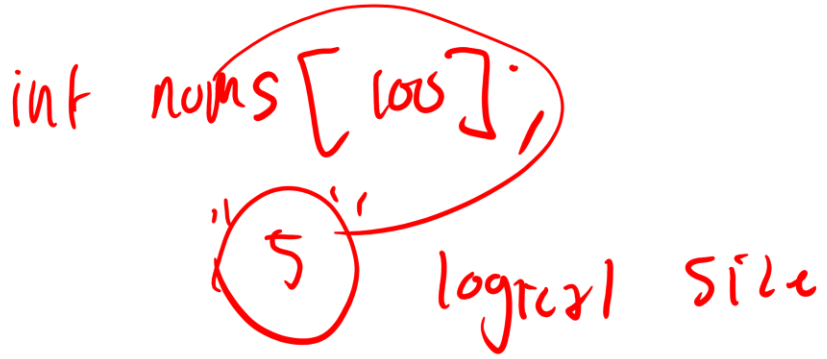
```
int nums[5][10][2];

for(          )
   for(          )
      for(          )
```

Multi-Dimensional Arrays /2

Beyond 3D arrays, it is often difficult to visualize them

Challenge



Write a function **`is_equal()`** that accepts three parameters: two integer arrays and an integer representing their size. You may assume both arrays have the same size. The function should return **1** if the two arrays are equal. This means that corresponding elements have the same values in the same positions. Otherwise, it should return **0**.

1, 2, 3

1, 3, 2

0, 1, 2 ... size-1

```
int is_equal(int arr1[], int arr2[], int size);
```

```
if ( arr1[i] != arr2[i] )
    return 0;
return 1;
```

Questions?