1. What is the physical structure of the array? How is the array stored in memory? Why is the index of the first element of the array [0] instead of [1]?
   a. In C, an array is stored as a **contiguous** block of memory. Each element of the array is placed sequentially in memory, with a fixed size per element.
      For an array:

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

      Assuming int takes 4 bytes, the memory layout looks like:

Index	Value	Memory Address (Example
array[0]	1	0x1000
array[1]	2	0x1004
array[2]	3	0x1008
array[3]	4	0x100C
array[4]	5	0x1010

      Each element is stored contiguously, with the next element's address being **current_address + sizeof(data_type)**.

   b. Arrays in C are stored in a **contiguous memory block** where:
      i. The **base address** is the address of the first element.
      ii. The next elements are stored at **sequential** memory locations.

      ```
      // Address of arr[i] = Base Address + (i * sizeof(type))
      // for arr[2];
      // Address = 1000 + (2 * 4) = 1008
      ```

   c. In C, an array name is a pointer to its first element. **arr[i]** is actually computed as:

      ```
      *(arr + i)  // Pointer arithmetic
      ```

      If **i** started from 1, we would have to adjust the calculation.we would need to subtract **sizeof(type)** every time, making calculations slightly less efficient.

2. What happens if an array overflows? Illustration
    a. An array overflow (or buffer overflow) occurs when a program writes beyond the allocated memory of an array. This leads to undefined behavior (UB), which can cause various problems such as:
        i. **Overwriting adjacent memory** (modifying other variables or program instructions).
        ii. **Segmentation fault (crash)** if it writes to a protected memory location.
        iii. **Security vulnerabilities**, allowing hackers to exploit the system.
    b. Illustration with Code :

```c
#include <stdio.h>

int main() {
    int x = 100; // Stored next to arr in memory
    int arr[7] = {1, 2, 3, 4, 5};

    printf("\n\nBefore overflow: x = %d\n", x);

    // Writing beyond the allocated memory of arr
    arr[7] = 999;  // Overflow: arr only has indexes 0 to 6!

    printf("After overflow: x = %d\n\n", x); // Unexpected behavior

    return 0;
}

```

Output:

```
Before overflow: x = 100
After overflow: x = 999
```

3. What can you learn from the following program?

```c
int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    printf("%x\n", array[1]);
    printf("%x\n", array[0]);
    printf("%x\n", array[-1]);

    printf("%x\n", array[4]);
    printf("%x\n", array[5]);
    return 0;
}
```

   a. This program provides key insights into **array indexing, memory access, and undefined behavior in C**.

      i. **Valid Array Access**
         1. array[0], array[1], and array[4] are valid accesses within the bounds of array[5]
         2. The program correctly prints their values in hexadecimal format (%x)

      ii. **Out-of-Bounds Access array[-1], array[5] (Undefined Behavior)**
         1. C does **not** perform bounds checking.
         2. Negative indexing accesses memory before the array, leading to undefined behavior
         3. It accesses memory beyond the allocated space, may print a random value, crash the program, or cause memory corruption.

      iii. **Undefined Behavior (UB) in C:**
         1. Print **garbage values, Corrupt adjacent memory,**
         2. Cause **segmentation faults** in some systems.
         3. Behave **differently on different compilers and architectures**.

      iv. **How to Avoid UB**:
         1. Always ensure array indices are within valid range
         2. Use **sizeof()** to determine array size dynamically

4. Briefly describe the storage and access of multi-dimensional arrays

   a. In C, multi-dimensional arrays (e.g. 2D arrays) are stored in row-major order, meaning that elements are stored in memory row by row. Visualization how these arrays are represented in memory and how to access them are given below:

b. Storage & Memory Representation of Multi-Dimensional Arrays:

i. Consider the declaration of a 2D array:

```
int arr[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

ii. **In memory**, it is stored **linearly** (as a flat array) in a single contiguous block. The elements are arranged row by row in memory. So, for a 2D array arr[2][3], the memory layout will look like this:

```
arr[0][0], arr[0][1], arr[0][2], arr[1][0], arr[1][1], arr[1][2]
```

c. Accessing Elements

i. One can access elements in a multi-dimensional array using the row and column indices where row and column index starts from 0.

For 2D array arr[2][3], syntax for accessing 1st row 2nd column element is:

```
int x = arr[0][1];  // Accessing the element in the second row, third column
                    //(value 2 for the previous example)
```

ii. the corresponding **linear index** can be calculated using following formulae:

```
// *(arr + i) + j  // This gives the address of arr[i][j]
// *(*(arr + i) + j)  // Dereferencing it gives the value at arr[i][j]
```

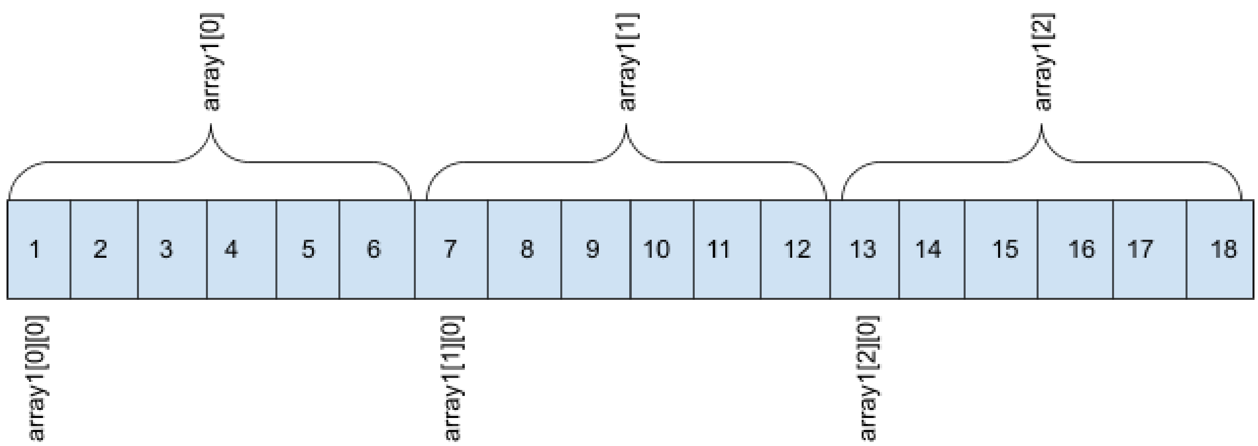
5. Draw some images to demonstrate the memory structure of array1/array2 and the output in the following program. Which can help a person know well how the memory stores an array

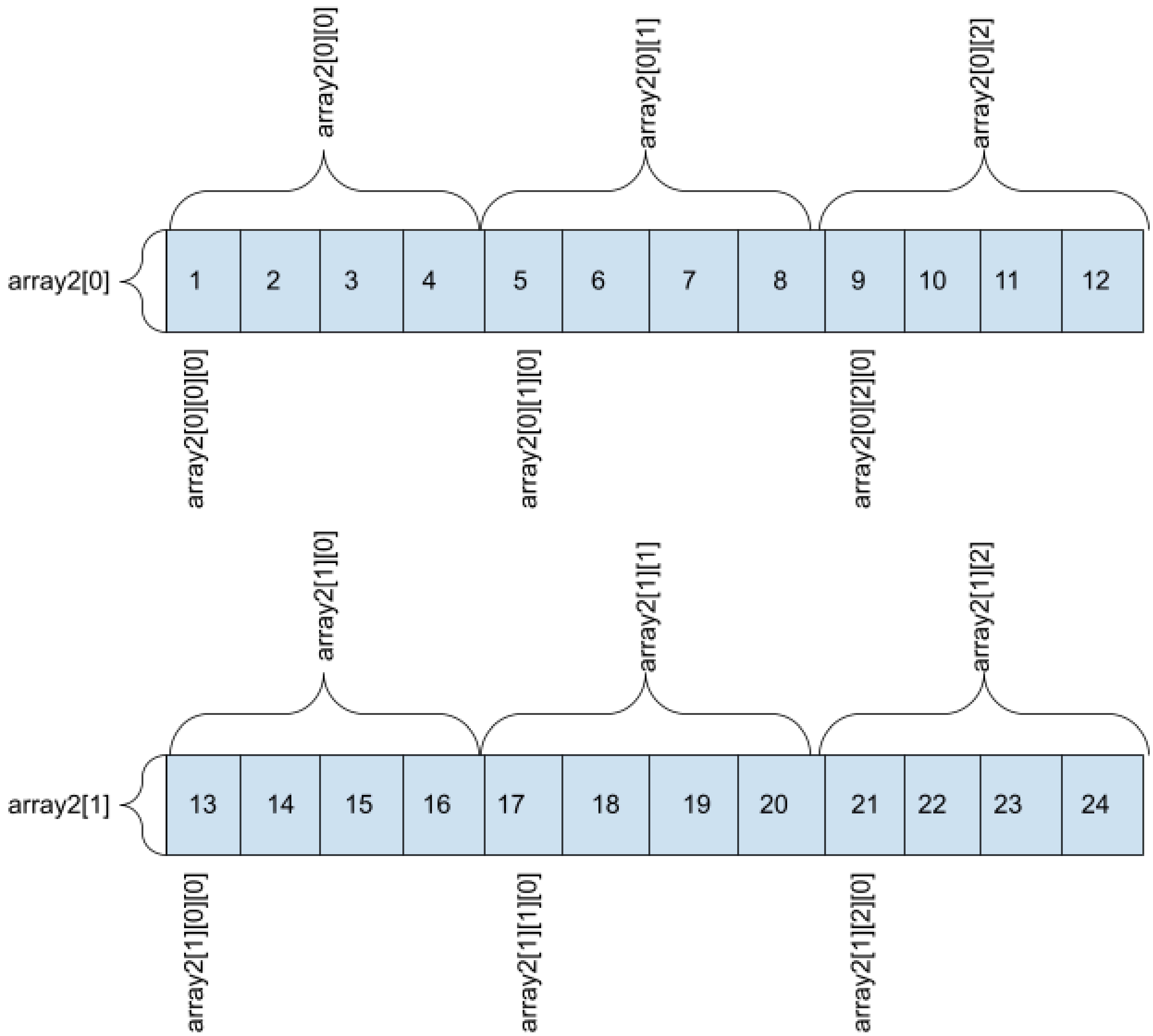
```c
int main(void)
{
    int array1[3][6] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};
    int array2[2][3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24};

    printf("%d %d\n", array1[1][6], array1[0][15]);         // 13 16
    printf("%d %d\n", array2[0][3][4], array2[0][2][8]);    // 17 17

    return 0;
}
```

a.





b. Output of the Programm:

   i. Applying formulae Address for 2D array **arr[i][j] =*(*(arr + i) + j)**
      assume base address is **100** in decimal
      array1[1][6]   = *((100 + 1*6*4) + 6*4) = *(148) = 13
      array1[0][15]   = *((100 + 0*6*4) + 15*4) = *(160) = 16

   ii. assume base address is **200** in decimal
      Applying formulae Address for 2D array **arr[i][j][k] = *(*(*(arr + i) + j) + k)**
      array2[0][3][4] = *(((200 + 0*3*4*4) + 3*4*4) + 4*4) = *(264) = 17
      array2[0][3][4] = *(((200 + 0*3*4*4) + 2*4*4) + 8*4) = *(264) = 17

c. Some key points which can help a person know well how the memory stores an array:

i. Single-Dimensional Array Memory Layout
    1. Stored in a continuous block of memory
    2. Address increments by sizeof(data type),

ii. Multi-Dimensional Array (Row-Major Storage)
    1. Memory is allocated row-wise in contiguous blocks

       To access an element:

```
Address = Base_Address + (row * num_cols + col) * sizeof(data type)
```

iii. Out-of-Bounds Access & Undefined Behavior