What is Data Structure?

To understand the concept of Data Structure, let's consider an example:

Example: Organizing Your Contacts on a Phone

Scenario:
You have 200 contacts saved on your phone, and you often need to call specific people. There are two ways to manage these contacts:

Method 1 – Unorganized List

- You save the contacts in random order as you add them:
  - **Contact 1: Akash**
  - **Contact 2: Karan**
  - **Contact 3: Rohan**
  - **Contact 4: Vishank**
  - **Contact 5: Aryan**
- **Problem:** If you want to call Vishank, you will have to scroll through the entire list to find his name. This takes a lot of time, especially as the number of contacts grows.

Method 2 – Organized Alphabetically
You arrange the contacts alphabetically:

- **A: Akash, Aryan**
- **K: Karan**
- **R: Rohan**
- **V: Vishank**
- **Benefit:** If you want to call Vishank, you can quickly jump to the "V" section and find his name. This makes searching faster and more efficient.

Which method is better?

- **Answer:** Method 2 is better because it organizes your contacts systematically, allowing you to find and access them quickly.

Example: Using a Single Notebook for Multiple Subjects

Scenario:

→ You are a student studying **five subjects**:

- **Maths-II**
- **DS Using Java**
- **Java-II**
- **FEE**
- **DBMS**

You need to take notes efficiently. There are two ways to do this:

Method 1 – Writing Randomly in One Notebook (Unorganized Approach)

- You write **Maths-II** notes on one page, then switch to **Java-II** randomly.
- Your **DS Using Java** notes are scattered in different places.
- When you need **JAVA-II** notes, you spend time **searching through the entire notebook**.
- **Problem:** It is **time-consuming and messy** to find specific subject notes.

Method 2 – Dividing the Notebook into Sections

- You **divide** the notebook into **five fixed sections**, one for each subject:
  - **Maths-II** → Pages **1-20**
  - **DS Using Java** → Pages **21-40**
  - **Java-II** → Pages **41-60**
  - **FEE** → Pages **61-80**
  - **DBMS** → Pages **81-100**
- **Benefit:** When you need **Java-II** notes, you go directly to the **Java-II section**, saving time.

Which Method is Better?

Method 2 is better because it follows a **structured data storage** approach.

How Does This Relate to Programming?

In programming, data structures help organize and retrieve data efficiently, just like organizing your phone contacts. For example:

- Arrays or Linked Lists store data in sequence.
- Binary Search Trees allow quick access by organizing data in sorted order.
- Hash Tables (or Maps) store data using keys for instant retrieval.

Conclusion: Learning data structures is crucial for writing programs that can handle large amounts of data efficiently and solve real-world problems effectively!

➢ What is Data Structure?

→ **A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.**

→ **A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data.**

→ There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

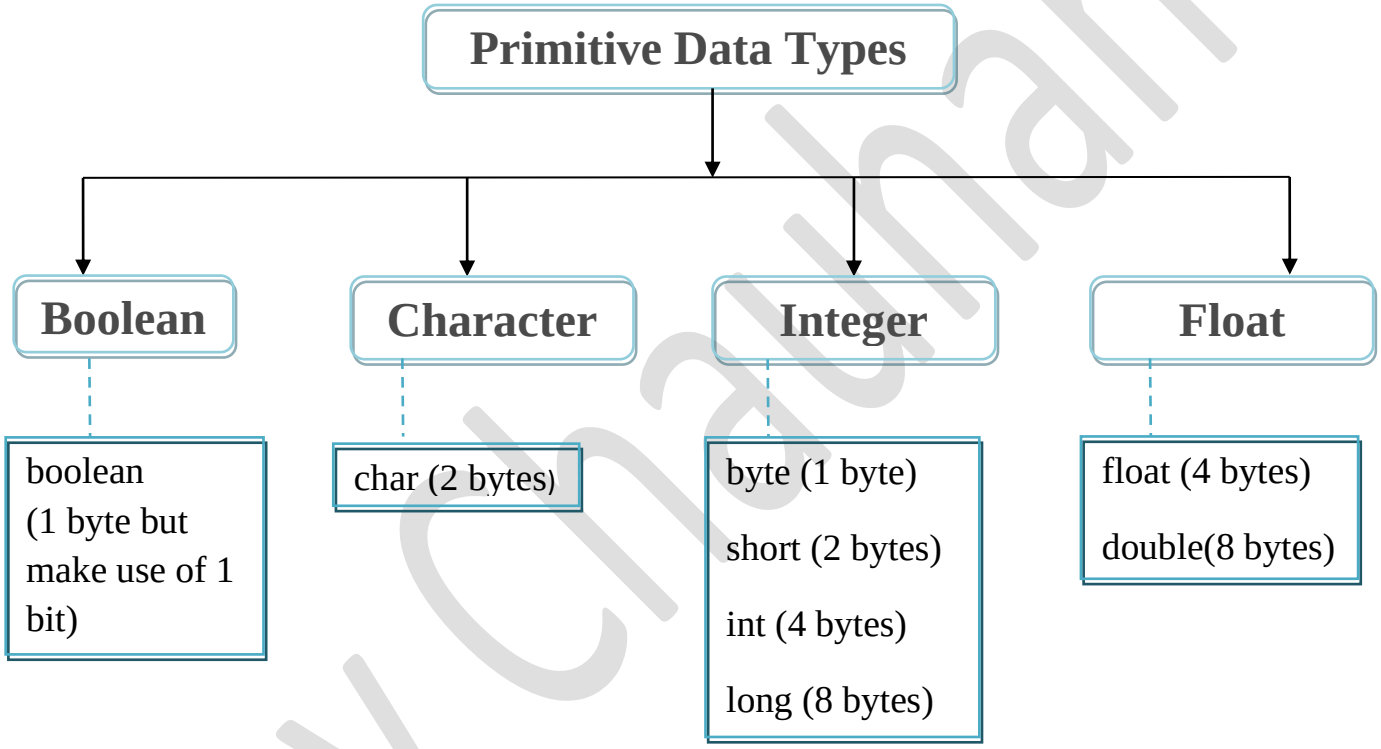
→ A data structure is a way to store data.

➢ Types of Data Structures

→ There are two types of data structures:

- **Primitive data structure**
- **Non-primitive data structure**

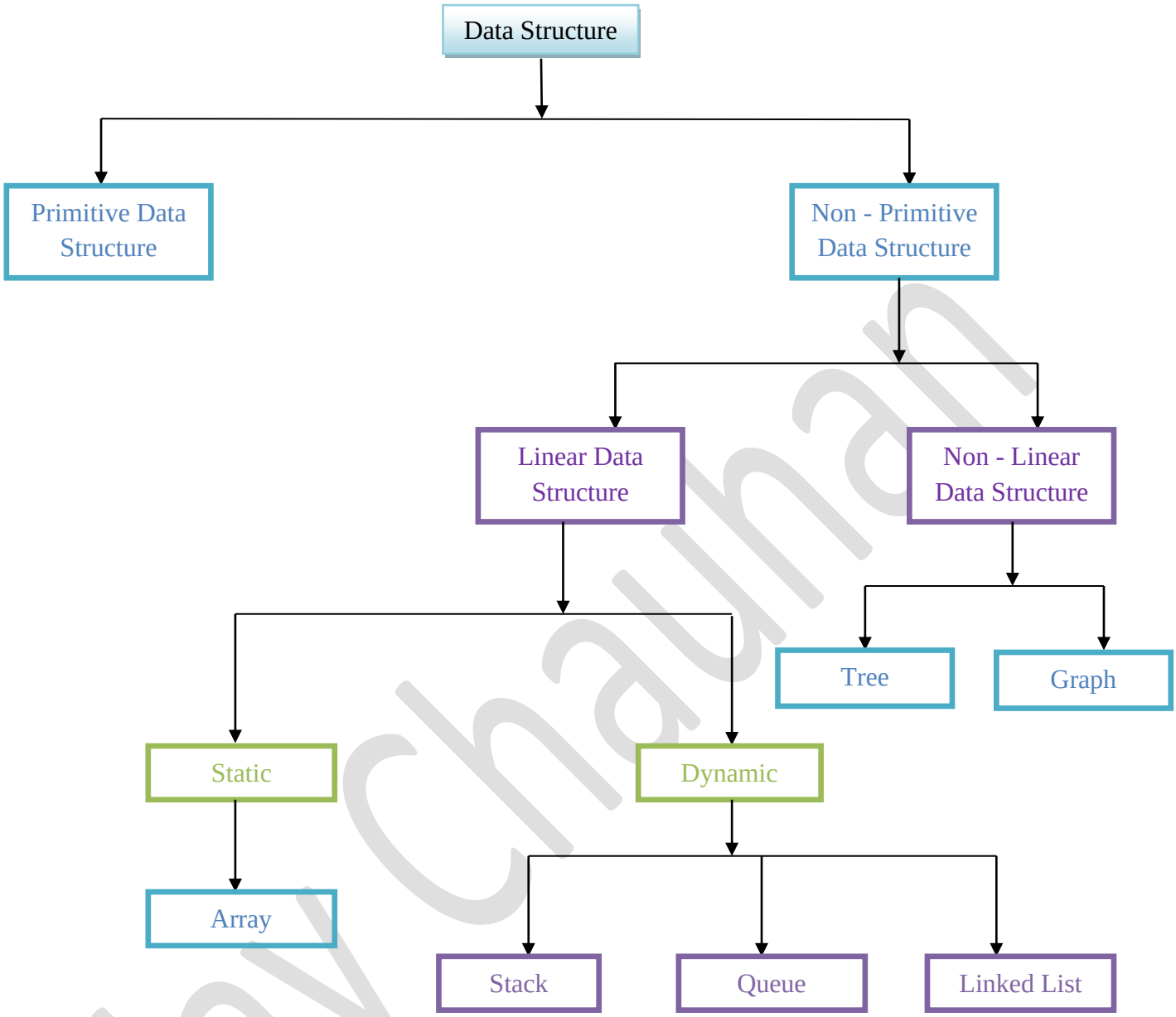
❖ **Primitive data structure :-** Primitive data structures are the basic building blocks of data. They store simple values and are directly supported by most programming languages.



❖ **Non-Primitive Data Structures :-** Non-primitive data structures are derived from primitive types and are more complex. They are categorized as:

➢ **Linear Data Structures:**
  - ➢ Arrays
  - ➢ Linked Lists
  - ➢ Stacks and Queues

➢ **Non-Linear Data Structures:**
  - ➢ Trees (e.g., Binary Tree)
  - ➢ Graphs



- **Linear Data Structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

Example: Array, Stack, Queue, Linked List, etc.

- **Static Data Structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

Example: array.

- **Dynamic Data Structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

Example: Queue, Stack, etc.

- **Non-Linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.

Examples: Trees and Graphs.

❖ Difference between data type and data structure

Data Types	Data Structures
A Data type is one of the forms of a variable to which the value can be assigned of a given type only. This value can be used throughout the program.	A Data structure is a collection of data of different data types. This collection of data can be represented using an object and can be used throughout the program.
The implementation of data type is known as an abstract implementation.	The implementation of data structure is known as a concrete implementation.
It can hold value but not data. Therefore, we can say that it is data-less.	It can hold multiple types of data within a single object.
In case of data type, a value can be assigned directly to the variables.	In the case of data structure, some operations are used to assign the data to the data structure object.
There is no problem in the time complexity.	When we deal with a data structure object, time complexity plays an important role.
The examples of data type are int, float, char.	The examples of data structure are stack, queue, tree, graph.

❖ Difference between Primitive and non-primitive data structure

Primitive data structure	Non-primitive data structure
Primitive data structure is a kind of data structure that stores the data of only one type.	Non-primitive data structure is a type of data structure that can store the data of more than one type.
Examples of primitive data structure are integer, character, float.	Examples of non-primitive data structure are Array, Linked list, stack.
Primitive data structure will contain some value, i.e., it cannot be NULL.	Non-primitive data structure can consist of a NULL value.
The size depends on the type of the data structure.	In case of non-primitive data structure, size is not fixed.

❖ Array Data Structure

→ Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

➢ Basic terminologies of Array

→ **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.

→ **Array element:** Elements are items stored in an array and can be accessed by their index.

→ **Array Length:** The length of an array is determined by the number of elements it can contain.

➢ Declaration of Array

```
// This array will store integer type element
int arr[];

// This array will store char type element
char arr[];

// This array will store float type element
float arr[];
```

➢ **Initialization of Array**

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[] = { 'a', 'b', 'c', 'd', 'e' };
float arr[] = { 1.4f, 2.0f, 24f, 5.0f, 0.0f };
```

❖ Why do we Need Arrays?

→ Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of records but what if the number of students becomes very large, it would be challenging to manipulate and maintain the data.

→ What it means is that, we can use normal variables (a, b, c, ..) when we have a small number of objects. But if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

```
int a = 10;
int b = 20;
int c = 30;
int d = 40;
int e = 50;
```

•
•
•

Multiple variable to store each values

Single Array to store all values

❖ Operations on Array

➢ **Initialization:** An array can be initialized with values at the time of declaration or later using an assignment statement.

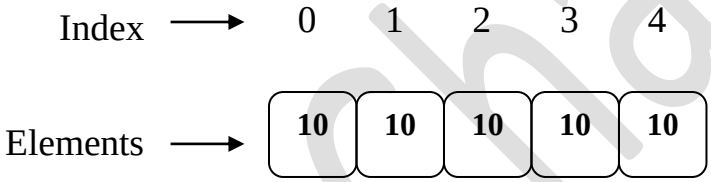
➢ **Accessing elements:** Elements in an array can be accessed by their index, which starts from 0 and goes up to the size of the array minus one.

➢ **Searching for elements:** Arrays can be searched for a specific element using linear search or binary search algorithms.

- **Sorting elements:** Elements in an array can be sorted in ascending or descending order using sorting algorithms like selection sort, bubble sort, or merge sort.
- **Inserting elements:** Elements can be inserted into an array at a specific position, but this operation requires shifting existing elements to make space.
- **Deleting elements:** Elements can be removed from an array by shifting the subsequent elements to fill the gap, reducing the effective size of the array.
- **Updating elements:** Elements in an array can be modified by assigning a new value to a specific index.
- **Traversing elements:** The elements in an array can be visited sequentially, either iteratively or recursively, to perform operations on each element.

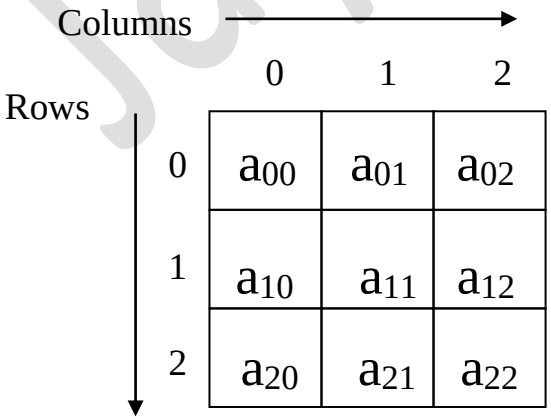
❖ Types of Arrays on the basis of Dimensions

1. One-dimensional Array(1-D Array): You can imagine a 1d array as a row, where elements are stored one after another.



2. Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

Two-Dimensional Array(2-D Array or Matrix): 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



➢ **Applications of Array Data Structure:**

→ Arrays mainly have advantages like random access and cache friendliness over other data structures that make them useful.

Below are some applications of arrays.

- **Storing and accessing data**: Arrays store elements in a specific order and allow constant-time O(1) access to any element.
- **Searching**: If data in array is sorted, we can search an item in O(log n) time. We can also find floor(), ceiling(), kth smallest, kth largest, etc efficiently.
- **Matrices**: Two-dimensional arrays are used for matrices in computations like graph algorithms and image processing.
- **Implementing other data structures:** Arrays are used as the underlying data structure for implementing stacks and queues.
- **Dynamic programming**: Dynamic programming algorithms often use arrays to store intermediate results of subproblems in order to solve a larger problem.
- **Data Buffers:** Arrays serve as data buffers and queues, temporarily storing incoming data like network packets, file streams, and database results before processing.

→ **Advantages of Array Data Structure:**

- **Efficient and Fast Access:** Arrays allow direct and efficient access to any element in the collection with constant access time, as the data is stored in contiguous memory locations.
- **Memory Efficiency:** Arrays store elements in contiguous memory, allowing efficient allocation in a single block and reducing memory fragmentation.
- **Versatility:** Arrays can be used to store a wide range of data types, including integers, floating-point numbers, characters, and even complex data structures such as objects and pointers.
- **Compatibility with hardware:** The array data structure is compatible with most hardware architectures, making it a versatile tool for programming in a wide range of environments.

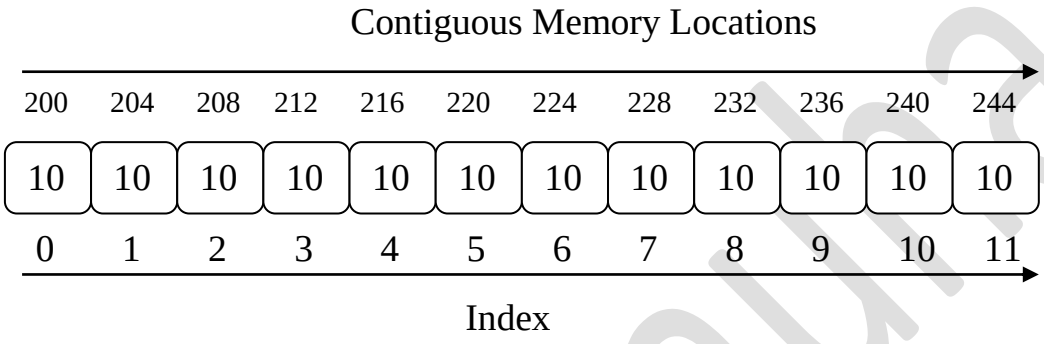
→ **Disadvantages of Array Data Structure:**

- **Fixed Size:** Arrays have a fixed size set at creation. Expanding an array requires creating a new one and copying elements, which is time-consuming and memory-intensive.
- **Memory Allocation Issues:** Allocating large arrays can cause memory exhaustion, leading to crashes, especially on systems with limited resources.
- **Insertion and Deletion Challenges:** Adding or removing elements requires shifting subsequent elements, making these operations inefficient.

- **Limited Data Type Support:** Arrays support only elements of the same type, limiting their use with complex data types.
- **Lack of Flexibility:** Fixed size and limited type support make arrays less adaptable than structures like linked lists or trees.

❖ Memory representation of Array

→ In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.



1. **Contiguous Memory Locations:**
   - Each element of the array is stored in a continuous block of memory.
   - The starting address of the array is 200 (as shown in the image), and each subsequent element is placed at the next available memory address.
2. **Indexing in Arrays:**
   - Arrays use **zero-based indexing**, meaning the first element is stored at index 0, the second at index 1, and so on.
   - In the given image, the array contains 12 elements, indexed from 0 to 11.
3. **Memory Address Calculation:**
   - Each element in the array occupies a fixed amount of memory.
   - If this is an int array, and each integer takes **4 bytes** of memory (typical for a 32-bit system), then:
     - The first element is at **200**.
     - The second element is at **204** (200 + 4).
     - The third element is at **208** (204 + 4), and so on.
4. **Formula for Address Calculation:**
   The memory address of an element at index i can be calculated as:

$$\boldsymbol{Address = Base\ Address + (Index \times Size\ of\ Data\ Type)}$$

   In this case, the base address is 200, and each element takes 4 bytes.

So, we can say that general formula to find the address of i^{th} element of the Array is,

$$\boldsymbol{Address = Base\ Address + (i - 1) \times Size\ of\ Single\ element}$$

$$\textbf{\textit{Number of elements in the matrix A}} = \textbf{\textit{Upper Bound}} - \textbf{\textit{Lower Bound}} + 1$$

Example 1:- Given the base address of an array A[0 …… 100] as 10 and the size of each element is 2 bytes in the memory, find the address of A[70].

Given Data:

Array: A[0 ... 100] (0-based indexing)

Base Address: 10

Size of Each Element: 2 bytes

Index to Find: A[70]

Formula : $\textbf{\textit{Address of A}}[i] = \textbf{\textit{Base Address}} + (\textbf{\textit{Index}} \times \textbf{\textit{Size of Data Type}})$

$$\begin{aligned} \textit{Address of A}\,[70] &= 10 + (70 \times 2) \\ &= 10 + 140 \\ &= 10 + 140 \\ &= 150 \end{aligned}$$

Example 2:- Given the base address of an array A[0 …… 100] as 10 and the size of each element is 2 bytes in the memory, find the address of 10th element.

Given Data:

Array: A[0 ... 100] (0-based indexing)

Base Address: 10

Size of Each Element: 2 bytes

Element to Find: 10th element

Formula: $\textbf{\textit{Address of A}}[i] = \textbf{\textit{Base Address}} + (i-1) \times \textbf{\textit{Size of Single element}}$

$$\begin{aligned} \textbf{\textit{Address of 10th element}} &= 10 + (10-1) \times 2 \\ &= 10 + (9 \times 2) \\ &= 10 + 18 \\ &= 10 + 18 \\ &= 28 \end{aligned}$$

Note that in the above program, we have calculated the address of the i^{th} element with reference to the 1st element (**index 0**) using the formula:

$$\textbf{\textit{Address of A}}[i] = \textbf{\textit{Base Address}} + (i-1) \times \textbf{\textit{Size of Single element}}$$

- But what if the array starts from an arbitrary index, such as **1300 to 1900** instead of **0-based indexing**?

In such cases, we need to modify the formula to accommodate the **lower bound (LB)** of the array:

$$\boldsymbol{Address = Base\ Address + (i - LB) \times Size\ of\ Single\ Element}$$

Example 3 :- Given the base address of an array A[1300 …… 1900] as 1020 and the size of each element is 2 bytes in the memory, find the address of A[1700].

Given Data:

Array: A[1300 ... 1900]

Base Address: 1020

Size of Each Element: 2 bytes

Element to Find: A[1700]

Lower Bound (LB): 1300

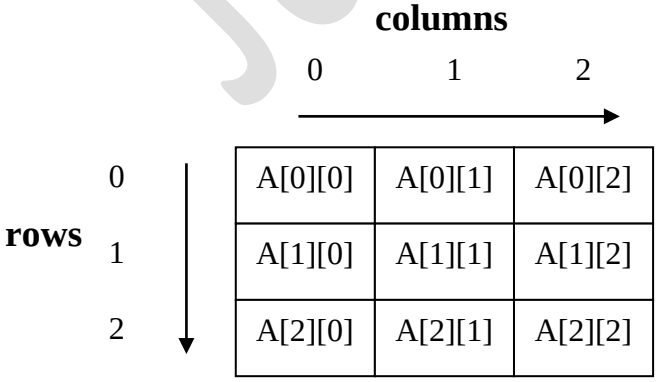
Formula for Address Calculation (for non-zero lower bound arrays):

$$\boldsymbol{Address = Base\ Address + (i - LB) \times Size\ of\ Single\ Element}$$

$$\begin{aligned} \boldsymbol{Address\ of\ A[1700]} &= \boldsymbol{1020 + (1700 - 1300) \times 2} \\ &= \boldsymbol{1020 + (400 \times 2)} \\ &= \boldsymbol{1020 + 800} \\ &= \boldsymbol{1820} \end{aligned}$$

Address of any element in the 2-D array:

→ **The 2-dimensional array can be defined as an array of arrays. The 2-Dimensional arrays are organized as matrices which can be represented as the collection of rows and columns as array[M][N] where M is the number of rows and N is the number of columns.**



To find the address of any element in a 2-Dimensional array there are the following two ways-

- **Row Major Order**
- **Column Major Order**

1. Row Major Order:

→ Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

→ The components of the first row are placed first in a 2D array, and then the components of the second row, and so on.

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

To find the address of the element using row-major order uses the following formula:

$$\boldsymbol{Address\ of\ A[i][j] = B + W * ((i - LR) * N + (j - LC))}$$

i = Row Subset of an element whose address to be found,
j = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

If the elements a, b, c, d, e, f, g, h, i are arranged in a 3×3 matrix and stored using row-major order, they will be stored in memory as follows:

Matrix Representation: A = $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

Row-Major Order Storage:

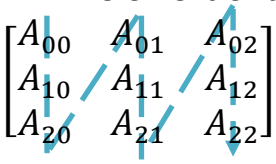
→ In row-major order, elements are stored row-wise, meaning that all elements of the first row are stored first, followed by the second row, and then the third row.

Memory Layout: a,b,c,d,e,f,g,h,i

2. Column Major Order:

→ If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order.

→ Each element of the initial column is placed first in a 2D array, followed by each element of the subsequent column, and so on.



To find the address of the element using column-major order use the following formula:

$$\boldsymbol{Address\ of\ A[i][j] = B + W * ((j - LC) * M + (i - LR))}$$

i = Row Subset of an element whose address to be found,
j = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.

If the elements a, b, c, d, e, f, g, h, i are arranged in a 3×3 matrix and stored using column-major order, they will be stored in memory as follows:

Matrix Representation: A = $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

Column-Major Order Storage in Memory:

In column-major order, the elements are stored as: a,d,g,b,e,h,c,f,i

Example-4 : Consider an integer array int a[3][4]. If the base address is 1050, find the address of the element a[2][2] using row-major and column-major representation of the array.

Given Data:
Array: a[3][4] (3 rows, 4 columns)
Base Address (B): 1050

Element to find: a[2][2] (Row index = 2, Column index = 2)
Size of each element (W) : Assume 4 bytes (since it's an int array)

1. Row-Major Order

Formula : $\boldsymbol{Address\ of\ A[i][j] = B + W * ((i - LR) * N + (j - LC))}$
where:

- i=2, j=2 (target element)
- LR=0, LC=0 (assuming **0-based indexing**)
- N=4 (total columns)

$$\begin{aligned} \boldsymbol{Address\ of\ A[i][j]} &= \boldsymbol{1050 + 4 * ((2-0) * 4 + (2-0))} \\ &= \boldsymbol{1050 + 4 * (8+2)} \\ &= \boldsymbol{1050 + 4 * 10} \\ &= \boldsymbol{1050 + 40} \\ &= \boldsymbol{1090} \end{aligned}$$

	0	1	2	3
0	1050	1054	1058	1062
1	1066	1070	1074	1078
2	1082	1086	**1090**	

2. Column-Major Order

Formula : $\boldsymbol{Address\ of\ A[i][j] = B + W * ((j - LC) * M + (i - LR))}$
where: m=3 (total rows)

$$\begin{aligned} \boldsymbol{Address\ of\ A[i][j]} &= \boldsymbol{1050 + 4 * ((2-0) * 3 + (2-0))} \\ &= \boldsymbol{1050 + 4 * (6+2)} \\ &= \boldsymbol{1050 + 4 * 8} \\ &= \boldsymbol{1050 + 32} \\ &= \boldsymbol{1082} \end{aligned}$$

	0	1	2	3
0	1050	1062	1074	
1	1054	1066	1078	
2	1058	1070	**1082**	

Example-5 : Given an array, arr[1……10][1……15] with base value 100 and the size of each element is 1 Byte in memory. Find the address of arr[8][6] with the help of row-major order.

Given:
Base address B = 100

Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found i = 8
Column Subset of an element whose address to be found j = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15

Formula : $\boldsymbol{Address\ of\ A[i][j] = B + W * ((i - LR) * N + (j - LC))}$

$$\begin{aligned} \boldsymbol{Address\ of\ A[8][6]} &= \mathbf{100 + 1 * ((8 - 1) * 15 + (6 - 1))} \\ &= \mathbf{100 + 1 * ((7) * 15 + (5))} \\ &= \mathbf{100 + 1 * (110)} \\ &= \mathbf{210} \end{aligned}$$

Example-6 : Given an array arr[1……10][1……15] with a base value of 100 and the size of each element is 1 Byte in memory find the address of arr[8][6] with the help of column-major order.

Given: **Base address B** = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found i = 8
Column Subset of an element whose address to be found j = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10

Formula : $\boldsymbol{Address\ of\ A[i][j] = B + W * ((j - LC) * M + (i - LR))}$

$$\begin{aligned} \boldsymbol{Address\ of\ A[8][6]} &= \mathbf{100 + 1 * ((6 - 1) * 10 + (8 - 1))} \\ &= \mathbf{100 + 1 * ((5) * 10 + (7))} \\ &= \mathbf{100 + 1 * (57)} \\ &= \mathbf{157} \end{aligned}$$

Example-7 : A 2-D array defined as A[r, c] where $2 \le r \le 6$, $3 \le c \le 9$, requires 4 bytes of memory for each element. If the array is stored in Column-major order, calculate the address of A[5,7] given the Base address as 5000.

Given Data:

- **Matrix size:**
  - **Row index:** $2 \le r \le 6 \rightarrow M = 6 - 2 + 1 = 5$ rows
  - **Column index:** $3 \le c \le 9 \rightarrow N = 9 - 3 + 1 = 7$ columns
- **Base address (B):** 5000
- **Element size (W):** 4 bytes
- **Lower bound row index (LR):** 2
- **Lower bound column index (LC):** 3
- **Target element (i, j):** A(5,7)
-

Formula : $\boldsymbol{Address(A[i][j]) = B + W \times ((j - LC) \times M + (i - LR))}$

$$\begin{aligned} Address(A[5][7]) &= 5000 + 4 \times ((7 - 3) \times 5 + (5 - 2)) \\ &= 5000 + 4 \times ((4) \times 5 + 3) \\ &= 5000 + 4 \times (20 + 3) \\ &= 5000 + 4 \times 23 \\ &= 5000 + 92 \\ &= 5092 \end{aligned}$$

Example-8 : Given a two-dimensional array M(3:8, 5:15) stored in column-major order with base address 500 and size of each element is 2 bytes, find the address of the element M(6, 10).

Given Data:

- **Matrix range:**
  - Row index: $3 \le r \le 8 \rightarrow M = 8 - 3 + 1 = 6$ rows
  - Column index: $5 \le c \le 15 \rightarrow N = 15 - 5 + 1 = 11$ columns
- **Base address (B):** 500
- **Element size (W):** 2 bytes
- **Lower bound row index (LR):** 3
- **Lower bound column index (LC):** 5
- **Target element (i, j):** M(6,10)
-

Formula : $\boldsymbol{Address(A[i][j]) = B + W \times ((j - LC) \times M + (i - LR))}$

$$\begin{aligned} Address(M[6][10]) &= 500 + 2 \times ((10 - 5) \times 6 + (6 - 3)) \\ &= 500 + 2 \times (5 \times 6 + 3) \\ &= 500 + 2 \times (30 + 3) \\ &= 500 + 2 \times 33 \\ &= 500 + 66 \\ &= 566 \end{aligned}$$

❖ Generalized Formulas for Row-Major and Column-Major Order with Explanation

→ When dealing with a 2D array stored in Row-Major Order (RMO) or Column-Major Order (CMO), we need to determine the row index and column index of a given k-th element (starting from base address).

1. Row-Major Order (RMO)

In **row-major order**, elements are stored **row-wise** in memory, meaning that all elements of a row are stored **before moving to the next row**.

Formulas for Row and Column Calculation in RMO

For an array **A[L1...U1][L2...U2]** (where L1 and L2 are lower bounds, and U1 and U2 are upper bounds):

- **Total Rows**:

$$R_{total} = U1 - L1 + 1$$

- **Total Columns**:

$$C_{total} = U2 - L2 + 1$$

- **Column Index Calculation**:

$$\boldsymbol{C = L2 + (k - 1)\ mod\ C_{total}}$$

- **Row Index Calculation**:

$$\mathbf{R = L1 + \left\lfloor \frac{K-1}{C_{total}} \right\rfloor}$$

Explanation:

- The **column index** is found using the modulus operator, which gives the remainder when the element position is divided by the total columns.
- The **row index** is found using integer division (floor function), which determines how many complete rows have been filled before reaching the **k-th element**.

2. Column-Major Order (CMO)

In **column-major order**, elements are stored **column-wise**, meaning that all elements of a column are stored **before moving to the next column**.

Formulas for Row and Column Calculation in CMO

For an array **A[L1...U1][L2...U2]**:

- **Total Rows**:

  $$R_{total} = U1 - L1 + 1$$

- **Total Columns**:

  $$C_{total} = U2 - L2 + 1$$

- **Row Index Calculation**:

  $$\boldsymbol{R = L1 + (k - 1) mod\ R_{total}}$$

- **Column Index Calculation**:

  $$\mathbf{C = L2 + \left\lceil \frac{(K-1)}{R_{total}} \right\rceil}$$

Explanation:

- The **row index** is found using the modulus operator, as elements are stored column-wise.
- The **column index** is determined by integer division, which tells us how many complete columns have been filled before reaching the **k-th element**.

- ✓ **RMO:** The row index is found using **integer division by total columns**, and the column index is found using **modulus by total columns**.
- ✓ **CMO:** The row index is found using **modulus by total rows**, and the column index is found using **integer division by total rows**.

Example-9 : Consider a 2D array A[-25...25][30....77], where each element occupies 2 bytes. Array is stored in ROW MAJOR ORDER. base address is 700. find row and column of 179th element.

Given:

- **Array:** A[−25...25][30....77]
- **Element size: 2 bytes**
- **Base address: 700**
- **Storage order: Row-major**
- **L1 = -25**
- **L2 = 30**
- **Number of rows(R_{total}) :** 25−(−25)+1= 51
- **Number of columns(C_{total}) :** 77−30+1=48
- **Total elements in the array:** 51×48=2448

Finding the Row and Column of the 179th Element:

Row-major order means elements are stored row-wise.

- **Row index calculation:**

$$\mathbf{R} = \mathbf{L1} + \left[\frac{K-1}{C_{total}}\right]$$

$$= -25 + \left[\frac{179-1}{48}\right]$$

$$= -25 + \left[\frac{178}{48}\right]$$

$$= -25 + 3\ (quotient)$$

$$= -22$$

- **Column Index Calculation**:

$$C = L2 + (k-1)\ mod\ C_{total}$$

$$= 30 + (179 - 1)\ mod\ 48$$

$= 30 + 178 \, mod \, 48$

$= 30 + 34$

$= 64$

Final Answer:

The row **and** column **of the** 179th element **are:** $(-22,64)$

Example-10 : Consider a 2-D array x with 11 rows and 4 columns, with each element storing a value equivalent to the product of row number and column number. The array is stored in row-major format. If the first element x[0][0] occupies the memory location with address 1000 and each element occupies only one memory location, which all locations(in decimal) will be holding a value of 10?

Step 1: Find all (i, j) pairs where x[i][j]=10

We solve: i×j=10

Possible integer solutions for (i, j) within the given **11 × 4** matrix:

1. (1,10) (out of bounds, since max column = 4) ❌
2. (2,5)(out of bounds, since max column = 4) ❌
3. (5,2) (Valid: 5≤10, 2≤3) ✅
4. (10,1) (Valid: 10≤10, 1≤3) ✅

So, valid positions are: (5,2) and (10,1)

Step 2: Calculate Memory Addresses (Row-Major Order)

The memory address formula is:

Formula : $Address\ of\ A[i][j] = B + W * ((i - LR) * N + (j - LC))$

- **Base Address** = 1000
- **Number of Columns** = 4
- **Element Size** = 1
- **Lower Bound of Rows and Columns = 0**

For x[5][2] :

$$Address\ of\ A[5][2] = 1000 + 1 \times ((5 - 0) \times 4 + (2 - 0))$$

= 1000 + (5 × 4 + 2)

= 1000 + (20+2)

= 1000 + 22

= 1022

For x[10][1]:

$$Address\ of\ A[10][1] = 1000 + 1 \times ((10 - 0) \times 4 + (1 - 0))$$

= 1000 + (10 × 4 + 1)

= 1000 + (40 + 1)

= 1000 + 41

= 1041

Java program to find the base address of the i^{th} element in a 1-D array

```
import java.util.Scanner;

class ArrayBaseAddress {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.println("Enter the size of the array:");
      int n = sc.nextInt();

      System.out.println("Enter the base address:");
      int baseAddress = sc.nextInt();
```

```
        System.out.println("Enter the size of a single element:");
        int elementSize = sc.nextInt();

        System.out.println("Enter the index of the element:");
        int i = sc.nextInt();

        if (i >= n) {
            System.out.println("Error: Index is out of bounds.");
        } else {
            int address = baseAddress + (i * elementSize);
            System.out.println("Address of the " + i + "th element: " +
address);
        }

    }
}
```

Java Program (for Position-based Address Calculation)

```
import java.util.Scanner;

class ArrayBaseAddress {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array:");
        int n = sc.nextInt();

        System.out.println("Enter the base address:");
        int baseAddress = sc.nextInt();

        System.out.println("Enter the size of a single element:");
```

```
        int elementSize = sc.nextInt();

        System.out.println("Enter the position of the element (1-based index):");
        int position = sc.nextInt();

        if (position < 1 || position > n) {
            System.out.println("Error: Position is out of bounds.");
        } else {
            int address = baseAddress + ((position - 1) * elementSize);
            System.out.println("Address of the element at position " + position + ": " + address);
        }


    }
}
```

Java program to compute the address of any element in a 2D array using Row-Major Order.

```
import java.util.Scanner;

class RowMajorOrder {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input array dimensions
        System.out.print("Enter number of rows (M): ");
        int M = sc.nextInt();
        System.out.print("Enter number of columns (N): ");
        int N = sc.nextInt();
```

```
        // Input base address and element size
        System.out.print("Enter base address: ");
        int B = sc.nextInt();
        System.out.print("Enter size of each element in bytes: ");
        int W = sc.nextInt();

        // Input lower bound indices
        System.out.print("Enter lower bound for rows (LR): ");
        int LR = sc.nextInt();
        System.out.print("Enter lower bound for columns (LC): ");
        int LC = sc.nextInt();

        // Input target element indices
        System.out.print("Enter row index of the target element (i): ");
        int i = sc.nextInt();
        System.out.print("Enter column index of the target element (j): ");
        int j = sc.nextInt();

        // Compute address using Row-Major formula
        int address = B + W * ((i - LR) * N + (j - LC));

        // Display result
        System.out.println("Memory address of A[" + i + "][" + j + "] in
row-major order: " + address);

    }
}
```

Java program to compute the address of any element in a 2D array using Column-Major Order.

```
import java.util.Scanner;

class ColumnMajorOrder {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input array dimensions
        System.out.print("Enter number of rows (M): ");
        int M = sc.nextInt();
        System.out.print("Enter number of columns (N): ");
        int N = sc.nextInt();

        // Input base address and element size
        System.out.print("Enter base address: ");
        int B = sc.nextInt();
        System.out.print("Enter size of each element in bytes: ");
        int W = sc.nextInt();

        // Input lower bound indices
        System.out.print("Enter lower bound for rows (LR): ");
        int LR = sc.nextInt();
        System.out.print("Enter lower bound for columns (LC): ");
        int LC = sc.nextInt();

        // Input target element indices
        System.out.print("Enter row index of the target element (i): ");
        int i = sc.nextInt();
        System.out.print("Enter column index of the target element (j): ");
        int j = sc.nextInt();
```

```
        // Compute address using Column-Major formula
        int address = B + W * ((j - LC) * M + (i - LR));

        // Display result
        System.out.println("Memory address of A[" + i + "][" + j + "] in
column-major order: " + address);

    }
}
```

Java program that calculates the memory address of an element in a 2D array using both Row-Major Order and Column-Major Order.

```
import java.util.Scanner;

class MemoryAddressCalculator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input dimensions
        System.out.print("Enter number of rows (r): ");
        int r = sc.nextInt();
        System.out.print("Enter number of columns (c): ");
        int c = sc.nextInt();

        // Input base address and size of each element
        System.out.print("Enter base address: ");
        int baseadd = sc.nextInt();
        System.out.print("Enter size of each element: ");
        int size = sc.nextInt();
```

```
        // Input lower bound indices
        System.out.print("Enter lower row index (LR): ");
        int LR = sc.nextInt();
        System.out.print("Enter lower column index (LC): ");
        int LC = sc.nextInt();

        // Input target element indices
        System.out.print("Enter row index (i): ");
        int i = sc.nextInt();
        System.out.print("Enter column index (j): ");
        int j = sc.nextInt();

        // Input method choice (1 = Row-Major, 2 = Column-Major)
        System.out.println("Choose method: 1 (Row-Major) or 2 (Column-
Major): ");
        int method = sc.nextInt();

        if (method == 1) {
            if (i >= LR && j >= LC && i < (LR + r) && j < (LC + c))
                {
                int ans = baseadd + size * ((i - LR) * c + (j - LC));
                System.out.println("Address of your given element is " + ans);
            }
else {
                System.out.println("You have entered incorrect values for i &
j.");
            }
        } else if (method == 2) {
            if (i >= LR && j >= LC && i < (LR + r) && j < (LC + c)) {
                int ans = baseadd + size * ((i - LR) + (j - LC) * r);
                System.out.println("Address of your given element is " + ans);
```

```
        } else {
            System.out.println("You have entered incorrect values for i & j.");
        }
    } else {
        System.out.println("Invalid method choice. Please enter 1 or 2.");
    }
  }
}
```