Table of Contents

---

1. Multidimensional Arrays

Definition

A multidimensional array is a **rectangular array** where all rows have the same number of columns. It's stored as a single contiguous block in memory.

Syntax & Declaration

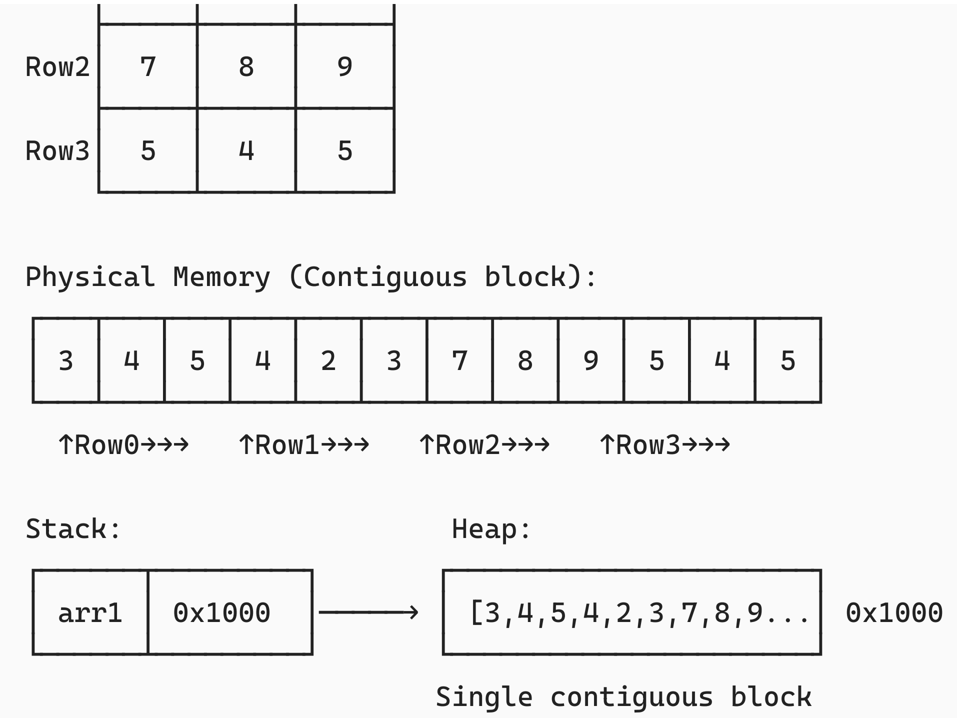
```
// Declaration - 2D array (4 rows × 3 columns)
int[,] arr = new int[4, 3];

// Declaration with initialization
int[,] arr1 = new int[4, 3]
{
    { 3, 4, 5 },   // Row 0
    { 4, 2, 3 },   // Row 1
    { 7, 8, 9 },   // Row 2
    { 5, 4, 5 }    // Row 3
};

// 3D array example
int[,,] cube = new int[2, 3, 4];  // 2 layers × 3 rows × 4 columns
```

Memory Representation

```
Logical View (4×3 array):
     Col0  Col1  Col2
    ┌─────┬─────┬─────┐
Row0│  3  │  4  │  5  │
    ├─────┼─────┼─────┤
Row1│  4  │  2  │  3  │
```



Row2	7	8	9
Row3	5	4	5

Physical Memory (Contiguous block):

3	4	5	4	2	3	7	8	9	5	4	5

↑Row0→→→ ↑Row1→→→ ↑Row2→→→ ↑Row3→→→

Stack:

arr1	0x1000

Heap:

[3,4,5,4,2,3,7,8,9...

0x1000

Single contiguous block

Accessing Elements

```
int[,] arr1 = new int[4, 3]
{
    { 3, 4, 5 },
    { 4, 2, 3 },
    { 7, 8, 9 },
    { 5, 4, 5 }
};

// Reading element at row 2, column 1
Console.WriteLine(arr1[2, 1]);  // Output: 8

// Modifying element at row 1, column 2
arr1[1, 2] = 4;
Console.WriteLine(arr1[1, 2]);  // Output: 4
```

Important Methods & Properties

```
int[,] matrix = new int[4, 3];

// Get number of dimensions
int dimensions = matrix.Rank;  // Returns 2

// Get length of specific dimension
int rows = matrix.GetLength(0);     // Returns 4 (dimension 0)
int cols = matrix.GetLength(1);     // Returns 3 (dimension 1)

// Get total number of elements
int total = matrix.Length;          // Returns 12 (4 × 3)

// Get upper bound of dimension (max index)
int maxRowIndex = matrix.GetUpperBound(0);  // Returns 3
int maxColIndex = matrix.GetUpperBound(1);  // Returns 2
```

Practical Example: Student Management System

```
// Scenario: Store student names across multiple tracks
// Each track has the SAME number of students

Console.WriteLine("Enter number of tracks:");
int numOfTracks = int.Parse(Console.ReadLine());

Console.WriteLine("Enter number of students per track:");
int numOfStudents = int.Parse(Console.ReadLine());

// Create 2D array: [tracks, students_per_track]
string[,] studentNames = new string[numOfTracks, numOfStudents];

// Input phase
for (int i = 0; i < studentNames.GetLength(0); i++)
{
    Console.WriteLine($"\n--- Track {i + 1} ---");

    for (int j = 0; j < studentNames.GetLength(1); j++)
    {
        Console.Write($"Enter name of student {j + 1}: ");
        studentNames[i, j] = Console.ReadLine();
    }
}

// Output phase
Console.WriteLine("\n========== STUDENT LIST ==========");
for (int i = 0; i < numOfTracks; i++)
{
    Console.WriteLine($"\n📚 Track {i + 1}:");
    Console.WriteLine("─────────────────────────");

    for (int j = 0; j < numOfStudents; j++)
    {
        Console.WriteLine($"  {j + 1}. {studentNames[i, j]}");
    }
}
```

Example Run:

```
Enter number of tracks: 2
Enter number of students per track: 3

--- Track 1 ---
Enter name of student 1: Ahmed
Enter name of student 2: Sara
Enter name of student 3: Mohamed

--- Track 2 ---
Enter name of student 1: Fatma
Enter name of student 2: Ali
Enter name of student 3: Nour

========== STUDENT LIST ==========

📚 Track 1:
─────────────────────────
  1. Ahmed
  2. Sara
  3. Mohamed

📚 Track 2:
─────────────────────────
  1. Fatma
  2. Ali
  3. Nour
```

Iteration Patterns

```
int[,] matrix = new int[3, 4];

// Pattern 1: Using GetLength (Recommended)
for (int i = 0; i < matrix.GetLength(0); i++)      // Rows
{
    for (int j = 0; j < matrix.GetLength(1); j++)  // Columns
    {
        Console.Write($"{matrix[i, j]} ");
    }
    Console.WriteLine();
}

// Pattern 2: Using GetUpperBound
for (int i = 0; i <= matrix.GetUpperBound(0); i++)
{
    for (int j = 0; j <= matrix.GetUpperBound(1); j++)
    {
        matrix[i, j] = i * j;
    }
}

// Pattern 3: Foreach (read-only access)
foreach (int value in matrix)
{
    Console.Write($"{value} ");
}
```

Advantages

1. **Simple syntax**: `arr[i, j]` is intuitive
2. **Memory efficient**: Stored as single contiguous block
3. **Cache-friendly**: Better CPU cache utilization
4. **Type-safe**: Strong compile-time checking
5. **Built-in bounds checking**: Prevents index out of range errors
6. **Easy to visualize**: Natural matrix/grid representation

Disadvantages

1. **Fixed rectangular shape**: All rows MUST have same column count
2. **Inflexible**: Cannot have rows with different lengths
3. **Harder to resize**: Cannot add/remove rows easily
4. **Memory waste**: If some rows need fewer columns, space is wasted
5. **Less intuitive for jagged data**: Not natural for variable-length rows

When to Use Multidimensional Arrays

Use when:

- Data forms a **perfect rectangle/grid** (e.g., chessboard, image pixels)
- All rows have **same number of columns**
- Need **maximum performance** (cache locality)
- Representing **mathematical matrices**
- Storing **tabular data** with uniform structure

Avoid when:

- Rows have **different lengths** (use jagged arrays)
- Need to **add/remove rows dynamically** (use List<List>)
- Data is **sparse** (many empty cells)

Real-world examples:

- 🎮 Game board (8×8 chess, 10×10 sudoku)
- 🖼️ Image pixels (width × height)
- 📊 Fixed-size spreadsheet
- 📅 Calendar (7 days × 5 weeks)
- 🎯 Seating chart (rows × seats per row)

By Abdullah Ali

Contact : +201012613453