2. Jagged Arrays (Array of Arrays)

Definition

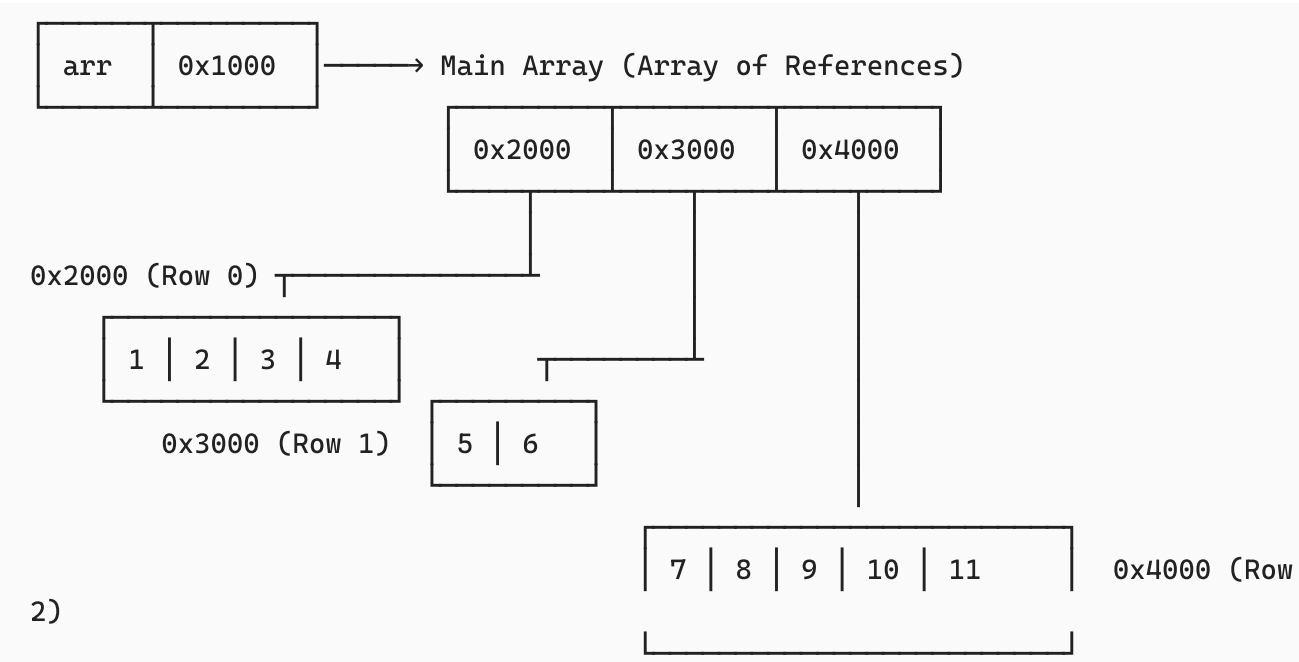
A jagged array is an **array of arrays** where each row can have a **different length**. It's stored as separate arrays in memory, with the main array holding references to sub-arrays.

Syntax & Declaration

```
// Declaration - creates array to hold 3 arrays
int[][] arr = new int[3][];

// Initialize each sub-array separately
arr[0] = new int[4];  // First row has 4 elements
arr[1] = new int[3];  // Second row has 3 elements
arr[2] = new int[4];  // Third row has 4 elements

// Declaration with initialization
int[][] arr2 = new int[][]
{
    new int[] { 1, 2, 3, 4 },      // Row 0: 4 elements
    new int[] { 5, 6 },            // Row 1: 2 elements
    new int[] { 7, 8, 9, 10, 11 }  // Row 2: 5 elements
};

// Shorthand syntax
int[][] arr3 =
{
    new int[] { 1, 2 },
    new int[] { 3, 4, 5 },
    new int[] { 6 }
};
```

Memory Representation

```
Logical View (Jagged):
Row 0: [1][2][3][4]
Row 1: [5][6]
Row 2: [7][8][9][10][11]

Physical Memory:
Stack:                    Heap:
```



Note: Each row is stored in a SEPARATE memory location!

Key Difference Visualization

MULTIDIMENSIONAL (int[,]):

One solid block:

1	2	3	4
5	6	7	8
9	10	11	12

All rows SAME length

JAGGED (int[][]):

Multiple separate arrays:

```
Main: [Ref1][Ref2][Ref3]
        ↓     ↓     ↓
  [1][2][3][4]
  [5][6]
  [7][8][9][10][11]
```

Each row DIFFERENT length ✓

Accessing Elements

```
int[][] arr = new int[3][];
arr[0] = new int[4];
arr[1] = new int[3];
arr[2] = new int[4];

// Accessing element: arr[row][column]
arr[0][0] = 4;                    // First row, first element
```

```
Console.WriteLine(arr[0][0]);     // Output: 4

// Get length of specific row
Console.WriteLine(arr[0].Length); // Output: 4
Console.WriteLine(arr[1].Length); // Output: 3

// Get number of rows
Console.WriteLine(arr.Length);    // Output: 3
```

Practical Example: Student Management (Variable Class Sizes)

```
// Scenario: Different tracks have DIFFERENT numbers of students
// This is where jagged arrays shine!

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

// Create jagged array
string[][] studentNames = new string[numOfTracks][];

// Input phase
for (int i = 0; i < numOfTracks; i++)
{
    Console.WriteLine($"\n--- Track {i + 1} ---");
    Console.Write("Enter number of students in this track: ");
    int numOfStudents = int.Parse(Console.ReadLine());

    // Initialize sub-array with specific size
    studentNames[i] = new string[numOfStudents];

    for (int j = 0; j < numOfStudents; 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 < studentNames.Length; i++)
{
    Console.WriteLine($"\n📚 Track {i + 1} ({studentNames[i].Length}
students):");
    Console.WriteLine("-------------------------");
```

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

Example Run:

```
Enter number of tracks: 3

--- Track 1 ---
Enter number of students in this track: 4
Enter name of student 1: Ahmed
Enter name of student 2: Sara
Enter name of student 3: Mohamed
Enter name of student 4: Fatma

--- Track 2 ---
Enter number of students in this track: 2
Enter name of student 1: Ali
Enter name of student 2: Nour

--- Track 3 ---
Enter number of students in this track: 5
Enter name of student 1: Omar
Enter name of student 2: Layla
Enter name of student 3: Youssef
Enter name of student 4: Hana
Enter name of student 5: Karim

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

📚 Track 1 (4 students):
------------------------
  1. Ahmed
  2. Sara
  3. Mohamed
  4. Fatma

📚 Track 2 (2 students):
------------------------
  1. Ali
  2. Nour

📚 Track 3 (5 students):
```

```
--------------------------
  1. Omar
  2. Layla
  3. Youssef
  4. Hana
  5. Karim
```

Iteration Patterns

```
int[][] jagged = new int[3][];
jagged[0] = new int[] { 1, 2, 3 };
jagged[1] = new int[] { 4, 5 };
jagged[2] = new int[] { 6, 7, 8, 9 };

// Pattern 1: Nested loops with Length (Recommended)
for (int i = 0; i < jagged.Length; i++)         // Rows
{
    for (int j = 0; j < jagged[i].Length; j++)  // Columns in row i
    {
        Console.Write($"{jagged[i][j]} ");
    }
    Console.WriteLine();
}

// Pattern 2: Foreach outer, for inner
foreach (int[] row in jagged)
{
    for (int j = 0; j < row.Length; j++)
    {
        Console.Write($"{row[j]} ");
    }
    Console.WriteLine();
}

// Pattern 3: Double foreach (read-only)
foreach (int[] row in jagged)
{
    foreach (int value in row)
    {
        Console.Write($"{value} ");
    }
    Console.WriteLine();
}

// Pattern 4: LINQ (advanced)
```

```
jagged.Select(row => string.Join(", ", row))
      .ToList()
      .ForEach(Console.WriteLine);
```

Advanced Operations

```
// Adding a new row (resize main array)
int[][] original = new int[2][];
original[0] = new int[] { 1, 2 };
original[1] = new int[] { 3, 4, 5 };

// Resize to add third row
Array.Resize(ref original, 3);
original[2] = new int[] { 6, 7 };

// Sorting rows by length
//result : original[0] -> original[2] -> original[1]
var sorted = original.OrderBy(row => row.Length).ToArray();

// Finding total elements
int totalElements = original.Sum(row => row.Length);

// Flattening jagged array
int[] flattened = original.SelectMany(row => row).ToArray();
// Result: [1, 2, 3, 4, 5, 6, 7]
```

Advantages

1. **Flexible row sizes**: Each row can have different length
2. **Memory efficient**: Only allocate space actually needed
3. **Dynamic growth**: Easier to add/remove rows
4. **Natural for hierarchical data**: Parent-child relationships
5. **Better for sparse data**: No wasted empty cells
6. **Common in real scenarios**: Most real data isn't perfectly rectangular

Disadvantages

1. **More complex syntax**: `arr[i][j]` with two separate indexers
2. **Slower access**: Extra pointer dereference per access
3. **More memory overhead**: Each sub-array has its own metadata
4. **Cache unfriendly**: Sub-arrays may not be contiguous in memory
5. **Manual initialization**: Must initialize each sub-array separately

6. **Null reference risk**: Sub-arrays can be null

When to Use Jagged Arrays

Use when:

- Rows have **different lengths**
- Data structure is **hierarchical** (parent-child)
- Need to **add/remove rows** dynamically
- Working with **sparse data** (many empty values)
- Memory efficiency matters for **irregular data**
- Real-world scenarios with **variable-sized collections**

Avoid when:

- All rows have **same length** (use multidimensional)
- Need **maximum performance** (cache locality matters)
- Simple **rectangular grid** is all you need

Real-world examples:

- 📚 Class roster (different class sizes)
- 🏢 Company hierarchy (departments with different team sizes)
- 📝 Paragraph structure (lines with different word counts)
- 🌳 Tree structures (nodes with varying children)
- 📊 Survey responses (respondents answer different questions)
- 📁 File system (folders with different file counts)

---

3. Comparison: Multidimensional vs Jagged

Side-by-Side Syntax Comparison

```
// ═══════════════════════════════════════════════════════
// DECLARATION
// ═══════════════════════════════════════════════════════
// Multidimensional (2D)
int[,] multi = new int[3, 4];

// Jagged
int[][] jagged = new int[3][];
jagged[0] = new int[4];
```

```
jagged[1] = new int[4];
jagged[2] = new int[4];

// ============================================================
// INITIALIZATION
// ============================================================
// Multidimensional
int[,] multi2 = { { 1, 2 }, { 3, 4 }, { 5, 6 } };

// Jagged
int[][] jagged2 =
{
    new int[] { 1, 2 },
    new int[] { 3, 4 },
    new int[] { 5, 6 }
};

// ============================================================
// ACCESS
// ============================================================
// Multidimensional: Single comma
int value1 = multi[1, 2];

// Jagged: Double brackets
int value2 = jagged[1][2];

// ============================================================
// GET DIMENSIONS
// ============================================================
// Multidimensional
int rows = multi.GetLength(0);
int cols = multi.GetLength(1);

// Jagged
int rows = jagged.Length;
int cols = jagged[0].Length;  // Only for first row!

// ============================================================
// ITERATION
// ============================================================
// Multidimensional
for (int i = 0; i < multi.GetLength(0); i++)
    for (int j = 0; j < multi.GetLength(1); j++)
        Console.Write(multi[i, j]);

// Jagged
```

```
for (int i = 0; i < jagged.Length; i++)
    for (int j = 0; j < jagged[i].Length; j++)
        Console.Write(jagged[i][j]);
```

Feature Comparison Table

Feature	Multidimensional `[,]`	Jagged `[][]`
Syntax	`arr[i, j]`	`arr[i][j]`
Row lengths	All rows SAME	Rows can differ
Memory layout	Single contiguous block	Multiple separate arrays
Memory efficiency	May waste space	Uses only needed space
Access speed	Faster (cache-friendly)	Slower (pointer indirection)
Initialization	Simple one-step	Must init each sub-array
Flexibility	Fixed rectangular	Dynamic variable-length
Null risk	No null sub-arrays	Sub-arrays can be null
Best for	Grids, matrices, tables	Hierarchies, varying data
Use cases	Game boards, pixels	Class rosters, trees

Performance Comparison

```
Operation               Multidimensional    Jagged
──────────────────────────────────────────────────
Element access          100%  ⚡⚡⚡⚡      85%  ⚡⚡⚡
Memory allocation       100%  ⚡⚡⚡⚡      110% ⚡⚡⚡
Iteration speed         100%  ⚡⚡⚡⚡      90%  ⚡⚡⚡
Cache efficiency        100%  ⚡⚡⚡⚡      60%  ⚡⚡
Flexibility             40%   ⚡           100% ⚡⚡⚡⚡
Memory waste (sparse)   High  ❌           Low  ✅
```

Decision Flowchart

```
Need 2D array?
    │
    ├─ All rows same length?
    │   ├─ Yes → Need max performance?
    │   │   ├─ Yes → Multidimensional [,]
    │   │   └─ No  → Either works (prefer [,] for simplicity)
    │   │
    │   └─ No (different lengths) → Jagged [][]
```

```
    └─ Need to add/remove rows dynamically?
        ├─ Yes → Jagged [][] or List<List<T>>
        └─ No  → Multidimensional [,]
```

Memory Usage Example

```
// Scenario: 3 tracks with 10, 5, and 8 students

// Multidimensional (must use max: 3×10 = 30 slots)
string[,] multi = new string[3, 10];
// Memory: 30 slots (5 wasted in row 2, 2 wasted in row 3)

// Jagged (exact fit: 10 + 5 + 8 = 23 slots)
string[][] jagged = new string[3][];
jagged[0] = new string[10];
jagged[1] = new string[5];
jagged[2] = new string[8];
// Memory: 23 slots + 3 sub-array references

// Waste calculation:
// Multidimensional: 30 slots (23% waste)
// Jagged: 23 slots + overhead (more efficient for this case)
```

Visual Comparison

```
Multidimensional [3, 4]:

┌───┬───┬───┬───┐
│ X │ X │ X │ X │ ← All rows MUST have 4 elements
├───┼───┼───┼───┤
│ X │ X │ X │ X │
├───┼───┼───┼───┤
│ X │ X │ X │ X │
└───┴───┴───┴───┘

Perfect rectangle ✓

Jagged [3][]:

┌───┬───┬───┬───┬───┐
│ X │ X │ X │ X │ X │ ← Row 0: 5 elements
└───┴───┴───┴───┴───┘

┌───┬───┐
│ X │ X │           ← Row 1: 2 elements
└───┴───┘

┌───┬───┬───┬───┐
```

Variable lengths ✓

---

4. Best Practices

Dos

1. **Choose the right array type**

```
// ✅ GOOD: Multidimensional for fixed grid
int[,] chessboard = new int[8, 8];

// ✅ GOOD: Jagged for variable data
string[][] departments = new string[5][];  // Different team sizes
```

2. **Always initialize jagged sub-arrays**

```
// ✅ GOOD
int[][] arr = new int[3][];
for (int i = 0; i < arr.Length; i++)
    arr[i] = new int[5];  // Initialize each row

// ❌ BAD: NullReferenceException!
int[][] bad = new int[3][];
bad[0][0] = 10;  // 💥 Crash!
```

3. **Use GetLength for multidimensional**

```
// ✅ GOOD
for (int i = 0; i < arr.GetLength(0); i++)
    for (int j = 0; j < arr.GetLength(1); j++)

// ❌ BAD: Won't work!
for (int i = 0; i < arr.Length; i++)  // Wrong for 2D!
```

4. **Cache array lengths in tight loops**

```
// ✅ GOOD: Cache lengths
int rows = matrix.GetLength(0);
int cols = matrix.GetLength(1);
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        Process(matrix[i, j]);
```

```
// ❌ BAD: Repeated calls
for (int i = 0; i < matrix.GetLength(0); i++)  // Called every iteration!
```

5. **Validate indices before access**

```
// ✅ GOOD
if (row >= 0 && row < jagged.Length &&
    col >= 0 && col < jagged[row].Length)
{
    return jagged[row][col];
}
```

6. **Use meaningful variable names**

```
// ✅ GOOD
for (int trackIndex = 0; trackIndex < tracks.Length; trackIndex++)
    for (int studentIndex = 0; studentIndex < tracks[trackIndex].Length;
studentIndex++)

// ❌ BAD
for (int i = 0; i < arr.Length; i++)
    for (int j = 0; j < arr[i].Length; j++)
```

Don'ts

1. **Don't use multidimensional for varying sizes**

```
// ❌ BAD: Wastes memory
string[,] students = new string[3, 100];  // What if track 1 has 10
students?

// ✅ GOOD
string[][] students = new string[3][];  // Each track gets exact size
```

2. **Don't forget to check for null in jagged**

```
// ❌ BAD
int total = jagged[0][0];  // 💥 If jagged[0] is null

// ✅ GOOD
if (jagged[0] != null)
    int total = jagged[0][0];
```

3. **Don't mix up syntax**

```
int[,] multi = new int[3, 4];
int[][] jagged = new int[3][];

// ❌ WRONG
```

```
int x = multi[1][2];    // Won't compile!
int y = jagged[1, 2];   // Won't compile!

// ✅ CORRECT
int x = multi[1, 2];    // Comma for multidimensional
int y = jagged[1][2];   // Double brackets for jagged
```

By Abdullah Ali

Contact : +201012613453