C Programming: 2D Arrays

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

January 16, 2026

Topics Covered

What are 2D Arrays?

- Array of arrays
- Organized in rows and columns (matrix form)
- All elements of same type
- Stored in contiguous memory (row-major order)
- Fixed size at declaration

Common Uses:

- Matrices (mathematical operations)
- Tables of data
- Game boards (chess, tic-tac-toe)
- Images (pixel data)
- Grids and maps

Indexing:

- `arr[row][col]` - both start at 0
- `arr[0][0]` - first element
- `arr[rows-1][cols-1]` - last element

2D Array Declaration and Syntax

Declaration:

```
data_type array_name[rows][cols];
```

Examples:

```
int matrix[3][4];       // 3 rows, 4 columns
float table[2][5];      // 2 rows, 5 columns
char grid[10][10];      // 10x10 grid
```

Memory Layout (Row-Major):

```
int arr[2][3] = {{1,2,3}, {4,5,6}};
// Memory: 1 2 3 4 5 6
// arr[0][0]=1, arr[0][1]=2, arr[0][2]=3
// arr[1][0]=4, arr[1][1]=5, arr[1][2]=6
```

Program 1: 2D Array Initialization

```c
#include <stdio.h>
int main() {
  int arr1[2][3] = {{1,2,3}, {4,5,6}};
  int arr2[2][3] = {1,2,3,4,5,6};
  int i, j;
  printf("Method 1 - Row by row:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr1[i][j]);
    }
    printf("\n");
  }
  printf("\nMethod 2 - Sequential:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr2[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Method 1 - Row by row:
1 2 3
4 5 6

Method 2 - Sequential:
1 2 3
4 5 6
```

Note:

- Both methods give same result
- Row-by-row is clearer
- Sequential fills row-major
- Nested loops for access

Program 2: Partial Initialization

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2}, {3}, {4,5,6}};
  int i, j;
  printf("Partially initialized:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Partially initialized:
1 2 0
3 0 0
4 5 6
```

Explanation:

- Row 0: 1, 2, rest 0
- Row 1: 3, rest 0
- Row 2: 4, 5, 6
- Unspecified elements = 0

Program 3: Zero Initialization

```c
#include <stdio.h>
int main() {
  int arr[3][4] = {0};
  int i, j;
  printf("Zero initialized 3x4:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Zero initialized 3x4:
0 0 0 0
0 0 0 0
0 0 0 0
```

Note:

- {0} sets all to zero
- Useful for counters, flags
- Better than uninitialized

Program 4: Reading 2D Array

```c
#include <stdio.h>
int main() {
  int arr[2][3];
  int input[6] = {10,20,30,40,50,60};
  int i, j, k = 0;
  printf("Enter 2x3 matrix:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      arr[i][j] = input[k++];
      printf("%d ", input[k-1]);
    }
    printf("\n");
  }
  printf("\nMatrix entered:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Enter 2x3 matrix:
10 20 30
40 50 60

Matrix entered:
10 20 30
40 50 60
```

Pattern:

- Nested loops for input
- Outer loop: rows
- Inner loop: columns
- Read row by row

Program 5: Display with Indices

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};
  int i, j;
  printf("Matrix with indices:\n");
  printf("     ");
  for (j = 0; j < 3; j++) {
    printf("col%d ", j);
  }
  printf("\n");
  for (i = 0; i < 3; i++) {
    printf("row%d: ", i);
    for (j = 0; j < 3; j++) {
      printf("%3d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Matrix with indices:
     col0 col1 col2
row0:   1    2    3
row1:   4    5    6
row2:   7    8    9
```

Note:

- Shows row and column labels
- Helpful for debugging
- %3d for alignment

Program 6: Sum of Each Row

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};
  int i, j, sum;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  printf("\nRow sums:\n");
  for (i = 0; i < 3; i++) {
    sum = 0;
    for (j = 0; j < 3; j++) {
      sum += arr[i][j];
    }
    printf("Row %d: %d\n", i, sum);
  }
  return 0;
}
```

Output:

```
Matrix:
1 2 3
4 5 6
7 8 9

Row sums:
Row 0: 6
Row 1: 15
Row 2: 24
```

Logic:

- For each row
- Initialize sum to 0
- Add all column elements
- Print row sum

Program 7: Sum of Each Column

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};
  int i, j, sum;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  printf("\nColumn sums:\n");
  for (j = 0; j < 3; j++) {
    sum = 0;
    for (i = 0; i < 3; i++) {
      sum += arr[i][j];
    }
    printf("Col %d: %d\n", j, sum);
  }
  return 0;
}
```

Output:

```
Matrix:
1 2 3
4 5 6
7 8 9

Column sums:
Col 0: 12
Col 1: 15
Col 2: 18
```

Logic:

- For each column
- Initialize sum to 0
- Add all row elements
- Note: loops swapped vs rows

Program 8: Maximum in Each Row

```c
#include <stdio.h>
int main() {
  int arr[3][4] = {{3,7,2,9},
                   {5,1,8,4},
                   {6,2,7,3}};
  int i, j, max;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  printf("\nMax in each row:\n");
  for (i = 0; i < 3; i++) {
    max = arr[i][0];
    for (j = 1; j < 4; j++) {
      if (arr[i][j] > max) {
        max = arr[i][j];
      }
    }
    printf("Row %d: %d\n", i, max);
  }
  return 0;
}
```

Output:

```
Matrix:
3 7 2 9
5 1 8 4
6 2 7 3

Max in each row:
Row 0: 9
Row 1: 8
Row 2: 7
```

Logic:

- For each row
- Assume first element is max
- Compare with rest
- Update if larger found

Program 9: Matrix Addition

```c
#include <stdio.h>
int main() {
  int a[2][2] = {{1,2}, {3,4}};
  int b[2][2] = {{5,6}, {7,8}};
  int sum[2][2];
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      sum[i][j] = a[i][j] + b[i][j];
    }
  }
  printf("Matrix A:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", a[i][j]);
    }
    printf("\n");
  }
  printf("\nMatrix B:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", b[i][j]);
    }
    printf("\n");
  }
  printf("\nA + B:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", sum[i][j]);
    }
    printf("\n");
  }
```

Output:

```
Matrix A:
1 2
3 4

Matrix B:
5 6
7 8

A + B:
6 8
10 12
```

Note:

- Element-wise addition
- Same dimensions required
- `sum[i][j] = a[i][j] + b[i][j]`

Program 10: Matrix Subtraction

```c
#include <stdio.h>
int main() {
  int a[2][3] = {{9,8,7}, {6,5,4}};
  int b[2][3] = {{1,2,3}, {4,5,6}};
  int diff[2][3];
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      diff[i][j] = a[i][j] - b[i][j];
    }
  }
  printf("A - B:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", diff[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
A - B:
8 6 4
2 0 -2
```

Note:

- Element-wise subtraction
- `diff[i][j] = a[i][j] - b[i][j]`
- Can produce negatives

Program 11: Matrix Multiplication

```c
#include <stdio.h>
int main() {
  int a[2][2] = {{1,2}, {3,4}};
  int b[2][2] = {{5,6}, {7,8}};
  int prod[2][2] = {0};
  int i, j, k;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      for (k = 0; k < 2; k++) {
        prod[i][j] += a[i][k]*b[k][j];
      }
    }
  }
  printf("A x B:\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", prod[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
A x B:
19 22
43 50
```

Formula:

- `prod[i][j] = sum(a[i][k]*b[k][j])`
- Triple nested loop
- $\text{Result}[0][0] = 1*5 + 2*7 = 19$
- $\text{Result}[0][1] = 1*6 + 2*8 = 22$

Program 12: Matrix Transpose

```c
#include <stdio.h>
int main() {
  int arr[3][2] = {{1,2}, {3,4}, {5,6}};
  int trans[2][3];
  int i, j;
  printf("Original (3x2):\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      trans[j][i] = arr[i][j];
    }
  }
  printf("\nTranspose (2x3):\n");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", trans[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Original (3x2):
1 2
3 4
5 6

Transpose (2x3):
1 3 5
2 4 6
```

Logic:

- Swap rows and columns
- `trans[j][i] = arr[i][j]`
- Dimensions reversed
- 3x2 becomes 2x3

Program 13: Sum of Diagonals

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};
  int i, j;
  int main_diag = 0, anti_diag = 0;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  for (i = 0; i < 3; i++) {
    main_diag += arr[i][i];
    anti_diag += arr[i][3-1-i];
  }
  printf("\nMain diagonal: %d\n",
         main_diag);
  printf("Anti diagonal: %d\n",
         anti_diag);
  return 0;
}
```

Output:

```
Matrix:
1 2 3
4 5 6
7 8 9

Main diagonal: 15
Anti diagonal: 15
```

Note:

- Main diagonal: i == j
- Elements: 1, 5, 9
- Anti diagonal: i + j == n-1
- Elements: 3, 5, 7

Program 14: Identity Matrix

```c
#include <stdio.h>
int main() {
  int identity[4][4] = {0};
  int i, j;
  for (i = 0; i < 4; i++) {
    identity[i][i] = 1;
  }
  printf("4x4 Identity Matrix:\n");
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      printf("%d ", identity[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
4x4 Identity Matrix:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Note:

- Initialize all to 0
- Set diagonal to 1
- `identity[i][i] = 1`
- Used in matrix math

Program 15: Search in 2D Array

```c
#include <stdio.h>
int main() {
  int arr[3][4] = {{10,20,30,40},
                   {15,25,35,45},
                   {12,22,32,42}};
  int target = 35;
  int found = 0;
  int i, j;
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
      if (arr[i][j] == target) {
        printf("Found %d at [%d][%d]\n",
               target, i, j);
        found = 1;
        break;
      }
    }
    if (found) break;
  }
  if (!found) {
    printf("%d not found\n", target);
  }
  return 0;
}
```

Output:

```
Found 35 at [1][2]
```

Logic:

- Nested loops to check all
- Compare each element
- Break from inner loop
- Break from outer if found

Program 16: Print Border Elements

```c
#include <stdio.h>
int main() {
  int arr[4][4] = {{1,2,3,4},
                   {5,6,7,8},
                   {9,10,11,12},
                   {13,14,15,16}};
  int i, j;
  printf("Matrix:\n");
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      printf("%2d ", arr[i][j]);
    }
    printf("\n");
  }
  printf("\nBorder elements:\n");
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      if (i==0 || i==3 || j==0 || j==3) {
        printf("%2d ", arr[i][j]);
      } else {
        printf("   ");
      }
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Matrix:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Border elements:
 1  2  3  4
 5        8
 9       12
13 14 15 16
```

Logic:

- First/last row
- First/last column

Program 17: Spiral Print

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};
  int top=0, bottom=2, left=0, right=2;
  int i;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    printf("%d %d %d\n",
           arr[i][0], arr[i][1], arr[i][2]);
  }
  printf("\nSpiral: ");
  while (top <= bottom && left <= right) {
    for (i=left; i<=right; i++)
      printf("%d ", arr[top][i]);
    top++;
    for (i=top; i<=bottom; i++)
      printf("%d ", arr[i][right]);
    right--;
    if (top <= bottom) {
      for (i=right; i>=left; i--)
        printf("%d ", arr[bottom][i]);
      bottom--;
    }
    if (left <= right) {
      for (i=bottom; i>=top; i--)
        printf("%d ", arr[i][left]);
      left++;
    }
  }
  printf("\n");
```

Output:

```
Matrix:
1 2 3
4 5 6
7 8 9

Spiral: 1 2 3 6 9 8 7 4 5
```

Logic:

- Right -¿ Down -¿ Left -¿ Up
- Shrink boundaries
- Continue until done

Program 18: Check Symmetric Matrix

```c
#include <stdio.h>
int main() {
  int arr[3][3] = {{1,2,3},
                   {2,4,5},
                   {3,5,6}};
  int i, j, symmetric = 1;
  printf("Matrix:\n");
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      if (arr[i][j] != arr[j][i]) {
        symmetric = 0;
        break;
      }
    }
    if (!symmetric) break;
  }
  printf("\n%s\n",
         symmetric ? "Symmetric" :
                     "Not symmetric");
  return 0;
}
```

Output:

```
Matrix:
1 2 3
2 4 5
3 5 6

Symmetric
```

Note:

- Symmetric: `arr[i][j] == arr[j][i]`
- Equal to its transpose
- Must be square matrix

Program 19: Upper Triangular Matrix

```c
#include <stdio.h>
int main() {
  int arr[4][4] = {0};
  int i, j, val = 1;
  for (i = 0; i < 4; i++) {
    for (j = i; j < 4; j++) {
      arr[i][j] = val++;
    }
  }
  printf("Upper Triangular:\n");
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      printf("%2d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Upper Triangular:
 1  2  3  4
 0  5  6  7
 0  0  8  9
 0  0  0 10
```

Note:

- Elements above diagonal
- `j >= i`
- Below diagonal: all zeros

Program 20: Lower Triangular Matrix

```c
#include <stdio.h>
int main() {
  int arr[4][4] = {0};
  int i, j, val = 1;
  for (i = 0; i < 4; i++) {
    for (j = 0; j <= i; j++) {
      arr[i][j] = val++;
    }
  }
  printf("Lower Triangular:\n");
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      printf("%2d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
Lower Triangular:
 1  0  0  0
 2  3  0  0
 4  5  6  0
 7  8  9 10
```

Note:

- Elements below diagonal
- `j <= i`
- Above diagonal: all zeros

2D Arrays - Summary

Key Points:

- Array of arrays: `arr[rows][cols]`
- Zero-based indexing for both dimensions
- Row-major memory layout
- Nested loops for traversal
- Initialization: row-by-row or sequential
- Matrix operations: add, subtract, multiply, transpose
- Row operations: sum, max, min per row
- Column operations: sum, max, min per column
- Diagonal operations: main and anti diagonal
- Search requires nested loops

Matrix Operations Reference

Operation	Formula/Note
Addition	`c[i][j] = a[i][j] + b[i][j]`
Subtraction	`c[i][j] = a[i][j] - b[i][j]`
Multiplication	`c[i][j] = sum(a[i][k] * b[k][j])`
Transpose	`trans[j][i] = arr[i][j]`
Main Diagonal	Elements where `i == j`
Anti Diagonal	Elements where `i + j == n-1`
Symmetric	`arr[i][j] == arr[j][i]`

Best Practices

1. **Always initialize** 2D arrays
2. **Use constants** for dimensions
3. **Check bounds** carefully (two dimensions!)
4. **Outer loop = rows**, inner loop = columns (usually)
5. **Pass dimensions** when using functions
6. **Visualize** the matrix structure
7. **Use meaningful names** - not just arr
8. **Format output** for readability
9. **Comment complex** traversal patterns
10. **Test with small** matrices first

Common Mistakes

1. **Index confusion**: `arr[j][i]` instead of `arr[i][j]`
2. **Wrong loop bounds**: Using rows for columns or vice versa
3. **Out of bounds**: Accessing `arr[rows][cols]`
4. **Dimension mismatch**: Adding matrices of different sizes
5. **Uninitialized arrays**: Reading before writing
6. **Wrong multiplication**: Element-wise instead of matrix multiply
7. **Forgetting break**: In nested search loops
8. **Swap confusion**: In transpose with same array
9. **Diagonal mistakes**: Off-by-one in anti-diagonal
10. **Memory layout**: Assuming column-major instead of row-major

Practice Exercises

Try these programs:

1. Rotate matrix 90 degrees clockwise
2. Check if matrix is diagonal
3. Find saddle point (min in row, max in column)
4. Print matrix in zigzag pattern
5. Interchange rows and columns
6. Check if two matrices are equal
7. Find determinant of 2x2 matrix
8. Convert matrix to 1D array
9. Find sum of all elements
10. Check if matrix is sparse (mostly zeros)