SNS COLLEGE OF TECHNOLOGY
(An Autonomous Institution)
COIMBATORE- 641 035

Arrays in C

Two-Dimensional Arrays

A **two-dimensional array** (2D array) is essentially an array of arrays. It is used to represent data in a matrix form or a table with rows and columns. Each element in a 2D array is accessed using two indices: one for the row and one for the column.

1. Declaring a Two-Dimensional Array

To declare a two-dimensional array, you specify the data type, the array name, the number of rows, and the number of columns.

Syntax:
data_type array_name[row_size][column_size];

- data_type: The type of the elements (e.g., int, float, char).
- array_name: The name of the array.
- row_size: The number of rows.
- column_size: The number of columns.

Example:
```
int matrix[3][4];  // 3 rows, 4 columns
```

In this example, matrix is a 2D array with 3 rows and 4 columns. It can hold a total of 3 * 4 = 12 elements.

2. Initializing a Two-Dimensional Array

You can initialize a 2D array either at the time of declaration or later in the program.

a. At Declaration

You can initialize a two-dimensional array with values by providing values for all the elements. The values are written row-wise inside curly braces {}.

```
int matrix[3][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

In this example, the array matrix is a 3x4 array, and its elements are initialized in a row-major order.

b. Partial Initialization

If you don't provide values for all the elements, the remaining elements are initialized to 0 (for numeric types like int).

```
int matrix[3][4] = {
   {1, 2, 3},  // 4th element will be 0
   {5, 6},     // 3rd and 4th elements will be 0
   {9}         // 2nd, 3rd, and 4th elements will be 0
};
```

c. Implicit Size Determination

You can omit the number of rows or columns, and the compiler will automatically determine the size based on the initializer list.

```
int matrix[][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

Here, the compiler determines that there are 3 rows, and 4 columns are explicitly specified.

3. Accessing Elements of a Two-Dimensional Array

To access elements in a two-dimensional array, you use the **row index** and **column index**.

array_name[row_index][column_index];

- row_index: The row number (starting from 0).
- column_index: The column number (starting from 0).

Example:

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

printf("Element at (1, 2): %d\n", matrix[1][2]);  // Accesses element at 2nd row, 3rd column (7)
```

4. Modifying Elements of a Two-Dimensional Array

You can modify elements of a 2D array by assigning new values using the row and column indices.

```
matrix[1][2] = 99;  // Modify the element at 2nd row, 3rd column to 99
```

5. Iterating Over a Two-Dimensional Array

To iterate over a 2D array, you can use two nested for loops: one for the rows and another for the columns.

Example:

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

for (int i = 0; i < 3; i++) {  // Loop through rows
    for (int j = 0; j < 4; j++) {  // Loop through columns
```

```
        printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
    }
}
```

This will print all elements of the matrix, row by row.

6. Calculating the Total Number of Elements in a Two-Dimensional Array

The total number of elements in a 2D array is the product of the number of rows and the number of columns. You can calculate it using the sizeof operator.

```
int rows = sizeof(matrix) / sizeof(matrix[0]);  // Number of rows
int columns = sizeof(matrix[0]) / sizeof(matrix[0][0]);  // Number of columns
```

Alternatively, if the dimensions are known, you can directly multiply:

```
int total_elements = 3 * 4;  // 3 rows * 4 columns = 12 elements
```

7. Example Program Using Two-Dimensional Arrays

Here's a simple program demonstrating the use of a 2D array in C:

```
#include <stdio.h>

int main() {
    // Declare and initialize a 3x4 matrix
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Print the original matrix
    printf("Original Matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
```

```
    // Modify an element (2nd row, 3rd column)
    matrix[1][2] = 99;

    // Print the modified matrix
    printf("\nModified Matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

yaml
Copy code

```
Original Matrix:
1 2 3 4
5 6 7 8
9 10 11 12

Modified Matrix:
1 2 3 4
5 6 99 8
9 10 11 12
```

8. Memory Layout of a Two-Dimensional Array

In C, a 2D array is stored in **row-major order**. This means that the elements of each row are stored in contiguous memory locations, followed by the elements of the next row, and so on.

Example:

Consider the following 2D array:

```
int matrix[2][3] = {
    {1, 2, 3},
```

```
    {4, 5, 6}
};
```

In memory, this array will be stored as:

csharp
Copy code
[1, 2, 3, 4, 5, 6]

Thus, the memory layout looks like a linear sequence where the rows are stored one after another.

9. Limitations of Two-Dimensional Arrays

1. **Fixed Size**: Like one-dimensional arrays, two-dimensional arrays in C have a fixed size. The number of rows and columns must be known at compile time for static arrays.
2. **No Bounds Checking**: C does not perform bounds checking, so accessing out-of-bounds indices can lead to undefined behavior.

10. Dynamic Two-Dimensional Arrays

If you need to create a 2D array where the number of rows and columns is determined at runtime, you can use **dynamic memory allocation** (via malloc or calloc).

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int rows = 3, cols = 4;
    int **matrix = (int **)malloc(rows * sizeof(int *));  // Allocate memory for rows

    for (int i = 0; i < rows; i++) {
        matrix[i] = (int *)malloc(cols * sizeof(int));  // Allocate memory for columns
in each row
    }

    // Initialize the matrix
    int value = 1;
    for (int i = 0; i < rows; i++) {
```

```
    for (int j = 0; j < cols; j++) {
        matrix[i][j] = value++;
    }
}

// Print the matrix
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}

// Free the allocated memory
for (int i = 0; i < rows; i++) {
    free(matrix[i]);
```