FUNDAMENTALS OF PROGRAMMING LANGUAGES

Unit IV: Arrays

Arrays in C Programming Language

```
int numbers[5] = {10, 20, 30, 40, 50};
char name[] = "C Arrays";
int matrix[3][3]; // 2D array
```

Prepared by: [Instructor Name]

Academic Year 2025

Table of Contents

1 Introduction to Arrays

2 One-Dimensional Arrays

3 1D Array Declaration and Initialization

4 Accessing and Updating 1D Arrays

5 Two-Dimensional Arrays

6 2D Array Declaration and Initialization

7 2D Array Memory Layout

8 Character Arrays and Strings

9 String Declaration and Initialization

10 String Operations & Handling Functions

11 Practical Examples & Memory Representation

12 Summary & Conclusion

Introduction to Arrays in C

Definition

An array in C is a collection of elements of the same data type stored in contiguous memory locations, accessed using an index.

Key Concepts

- Fixed size determined at declaration
- Zero-indexed (first element at index 0)
- Contiguous memory allocation
- Elements must share the same data type
- Can be 1D, 2D, or multidimensional

Array Visualization

Index

0	1	2	3	4
10	20	30	40	50

Values

Example Declaration

```c
#include <stdio.h>

int main() {
    // Array declaration & initialization
    int numbers[5] = {10, 20, 30, 40, 50};

    // Accessing array elements
    printf("%d", numbers[2]); // Outputs: 30

    return 0;
}
```

Applications of Arrays

- Data storage & manipulation
- Implementing matrices & tables
- Sorting & searching algorithms
- Statistical computations

One-Dimensional Arrays

Definition

A one-dimensional array is a linear collection of elements of the same data type, arranged sequentially in memory and accessed using a single index.

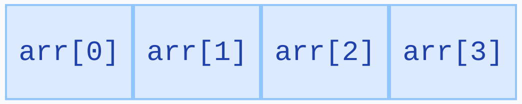
General Syntax

```
data_type array_name[array_size];
```

Where:

- **data_type**: Type of all elements (int, float, char, etc.)
- **array_name**: Identifier for the array
- **array_size**: Number of elements (fixed at compile-time)

Memory Representation



Base Address → Contiguous Memory Locations

Example Code

```c
#include <stdio.h>

int main() {
    // Declare an array of 5 integers
    int numbers[5];

    // Initialize array elements
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

    // Access and print third element
    printf("%d", numbers[2]); // Outputs: 30

    return 0;
}
```

Key Concepts of 1D Arrays

- ✓ Zero-Indexed
  First element is at index 0
- Contiguous Memory
  Elements stored adjacent in memory
- ⚠ Boundary Checking
  No automatic bounds checking
- Fixed Size
  Size must be defined at declaration
- Homogeneous
  All elements must be same data type
- Array Name
  Represents address of first element

1D Array Declaration and Initialization

Declaration Syntax

Declaration reserves memory for the array without assigning values:

```
data_type array_name[array_size];
```

Examples:

- `int marks[5];` - Array of 5 integers
- `float prices[100];` - Array of 100 floats
- `char letters[26];` - Array of 26 characters

Initialization Methods

Arrays can be initialized in several ways:

- **At declaration time:** `int nums[5] = {10, 20, 30, 40, 50};`
- **Partial initialization:** `int nums[5] = {10, 20};` (rest are 0)
- **Omitting size:** `int nums[] = {10, 20, 30};` (size is 3)
- **After declaration:** Using loops or individual assignments

Visualization of Initialization

int numbers[5] = {10, 20, 30, 40, 50};

0	1	2	3	4
10	20	30	40	50

int numbers[5] = {10, 20}; (Partial initialization)

0	1	2	3	4
10	20	0	0	0

Initialization Examples

```
#include <stdio.h>

int main() {
    // Complete initialization
    int marks[5] = {95, 88, 76, 90, 79};

    // Partial initialization (rest set to 0)
    int counts[5] = {1, 2};

    // Size determined by elements
    int scores[] = {98, 87, 92};

    // Individual element initialization
    int values[3];
    values[0] = 5;
    values[1] = 10;
    values[2] = 15;

    return 0;
}
```

Important Notes

- Cannot initialize during declaration with values not known at compile time
- Uninitialized array elements have garbage values
- Size must be a constant integer expression

Accessing and Updating 1D Array Elements

Accessing Array Elements

Array elements are accessed using the index within square brackets:

- Syntax: `array_name[index]`
- Arrays are zero-indexed (first element at index 0)
- Valid indices range from 0 to size-1
- Example: `value = arr[2];` accesses the 3rd element

Updating Array Elements

Elements can be modified using assignment operators:

- Syntax: `array_name[index] = new_value;`
- Example: `arr[3] = 45;` updates the 4th element
- Array elements can be updated multiple times

Array Access Visualization

0	1	2	3	4
25	32	17	94	63

arr[2] = 17

Array Access Example

```c
#include <stdio.h>

int main() {
    int arr[5] = {25, 32, 17, 94, 63};

    // Accessing array elements
    printf("Third element: %d\n", arr[2]);

    // Updating array elements
    arr[3] = 45;
    printf("Updated fourth element: %d\n", arr[3]);

    return 0;
}
```

Array Traversal Using Loops

For Loop Traversal

```c
// Forward traversal
for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
}

// Backward traversal
for (int i = size-1; i >= 0; i--) {
    printf("%d ", arr[i]);
}
```

Common Pitfalls

- ⚠ Accessing out-of-bounds indices (array[size])
- ⚠ Using negative indices
- ⚠ Forgetting array indices start at 0
- ⚠ Not checking array bounds in loops

Two-Dimensional Arrays

Definition

A two-dimensional array in C is an array of arrays - essentially a table or matrix with rows and columns, storing elements of the same data type.

Key Concepts

- Represented as a matrix with rows and columns
- Elements accessed using two indices: row and column
- Stored in row-major order in memory (by default)
- Size determined by number of rows × number of columns
- Useful for tabular data, matrices, and grids

2D Array Visualization

	Col 0	Col 1	Col 2
Row 0	10	20	30
Row 1	40	50	60

Matrix[2][3] - 2 rows, 3 columns

Syntax & Example

```
// Declaration syntax
type array_name[rows][columns];

// Example initialization
int matrix[2][3] = {
    {10, 20, 30},   // Row 0
    {40, 50, 60}    // Row 1
};

// Access element at row 1, column 2
int value = matrix[1][2];  // value = 60
```

Applications of 2D Arrays

- Spreadsheets & tables
- Game boards (chess, tic-tac-toe)
- Digital image processing
- Matrix operations

2D Array Declaration and Initialization

Declaration Syntax

A 2D array in C is declared by specifying the type followed by the array name and two sets of square brackets:

```
type array_name[rows][columns];
```

For example:

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

Initialization Methods

- At declaration:

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

- Row-by-row initialization:

```
int arr[2][3] = {1, 2, 3, 4, 5, 6}; // Same result
```

- Partial initialization:

```
int arr[2][3] = {{1, 2}, {4}}; // Rest filled with 0
```

Matrix Representation

int matrix[2][3]

0,0	0,1	0,2
1,0	1,1	1,2

1	2	3
4	5	6

Values

Nested Loop Traversal

```
#include <stdio.h>

int main() {
    int matrix[2][3] = {{1, 2, 3},
                        {4, 5, 6}};

    // Traversing all elements
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n"); // New line after each row
    }

    return 0;
}
```

Output:

```
1 2 3 4 5 6
```

Accessing Elements

Direct Access
matrix[1][2] = 6;

Updating Elements
matrix[0][1] = 10;

Traversal Pattern
Row-by-row, column-by-column

Memory Layout
Row-major ordering in C

2D Array Memory Layout

Row-Major Order Storage

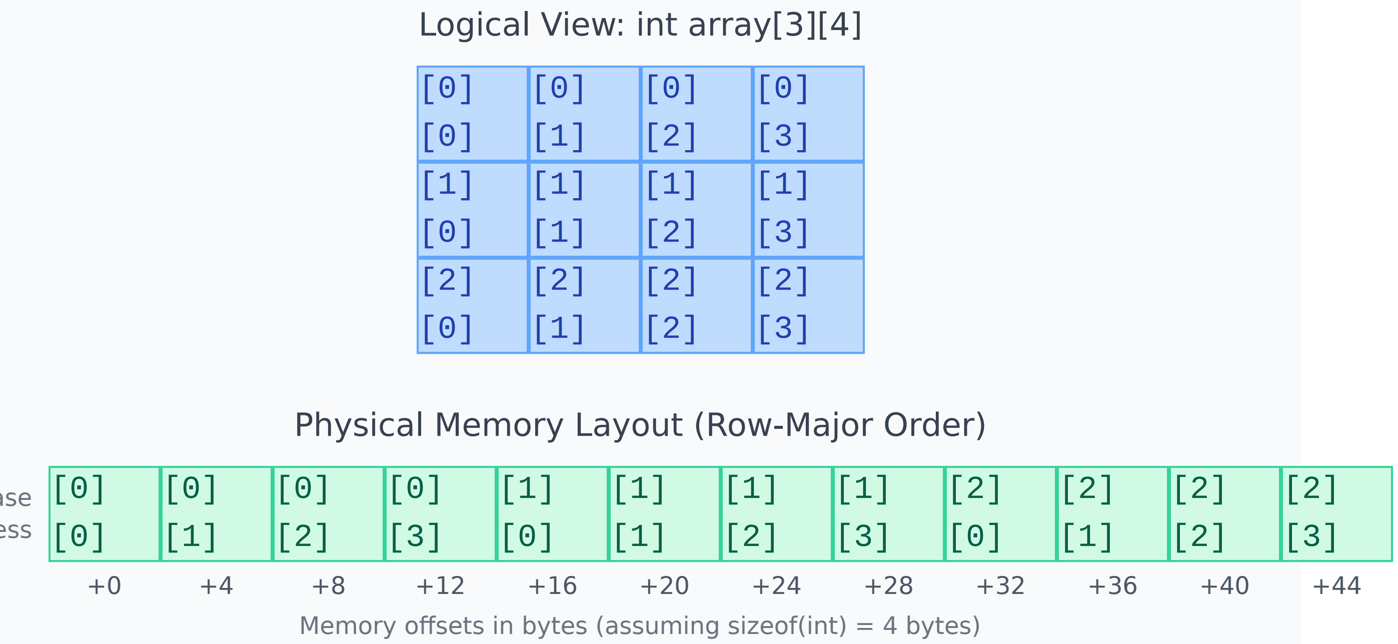
In C, 2D arrays are stored in **row-major order**, which means elements are stored row by row in contiguous memory locations.

- Each row is stored sequentially in memory
- First all elements of row 0, then row 1, etc.
- Efficient for row-wise operations

Address Calculation

```
For array[rows][cols]:
Address of array[i][j] =
  base_address +
  (i × cols + j) × sizeof(datatype)
```

2D Array Memory Visualization



Logical View: int array[3][4]

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

Physical Memory Layout (Row-Major Order)

Base Address	[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]	[2][1]	[2][2]	[2][3]
	+0	+4	+8	+12	+16	+20	+24	+28	+32	+36	+40	+44

Memory offsets in bytes (assuming sizeof(int) = 4 bytes)

Example: Accessing 2D Array Elements

```c
#include <stdio.h>

int main() {
    int matrix[3][4] = {
        {10, 20, 30, 40},
        {50, 60, 70, 80},
        {90, 100, 110, 120}
    };

    // Accessing element matrix[1][2] = 70
    printf("%d", matrix[1][2]);

    return 0;
}
```

Character Arrays and Strings in C

Definition

A string in C is an array of characters terminated by a null character ('\0'). C doesn't have a built-in string data type, instead it uses character arrays to store and manipulate text.

Key Concepts

- Null terminator ('\0') marks the end of the string
- String length is the number of characters excluding '\0'
- String literals are enclosed in double quotes
- Array size must accommodate the null terminator
- Strings can be accessed character-by-character

String Visualization

char str[] = "Hello";

0	1	2	3	4	5
'H'	'e'	'l'	'l'	'o'	**'\0'**

Null terminator at the end

String Declaration

```
// Method 1: Using string literals
char str1[] = "Hello";

// Method 2: Character array with null
char str2[] = {'H','e','l','l','o','\0'};

// Method 3: Fixed size array
char str3[10] = "Hello";

// Access characters
printf("%c", str1[1]); // Outputs: 'e'
```

String Initialization & Operations

- Reading with scanf("%s", str) or fgets(str, size, stdin)
- Writing with printf("%s", str) or puts(str)
- Comparing with strcmp(s1, s2)
- Concatenating with strcat(dest, src)

String Operations & Handling Functions

Common String Functions

The `<string.h>` header provides various functions for string manipulation:

`strlen(str)`	Returns length of string (excluding null)
`strcpy(dest, src)`	Copies source string to destination
`strcat(dest, src)`	Appends source string to destination
`strcmp(s1, s2)`	Compares two strings (returns 0 if equal)
`strncpy(dest, src, n)`	Copies up to n characters

String I/O Functions

Reading strings:

`scanf("%s", str)` - reads until whitespace

`scanf("%[^\n]s", str)` - reads line with spaces

`fgets(str, size, stdin)` - safer, reads with size limit

Writing strings:

`printf("%s", str)` - prints string

`puts(str)` - prints string with newline

String Functions Example

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";
    char str3[40];

    // Get string length
    printf("Length: %lu\n", strlen(str1));

    // Copy string
    strcpy(str3, str1);

    // Concatenate strings
    strcat(str3, " ");
    strcat(str3, str2);

    // Compare strings
    if(strcmp(str1, str2) != 0) {
        printf("Strings are different\n");
    }

    return 0;
}
```

String Function Visualization

strlen("Hello") = 5

H	e	l	l	o	\0

strcat(str1, str2)

H	e	l	l	o	W	o	r	l	d	\0

Practical Applications

User input validation

Text processing

Password verification

Data parsing & manipulation

Summary & Conclusion

Key Takeaways

- ✓ Arrays provide efficient storage for collections of similar data types
- ✓ One-dimensional arrays store linear sequences of elements
- ✓ Two-dimensional arrays represent tabular data and matrices
- ✓ Character arrays with null terminators represent strings
- ✓ Standard library provides rich string manipulation functions

Practical Applications

Numerical Analysis

Statistical computations, matrices for scientific applications

Data Tables

Storing and manipulating tabular data efficiently

Text Processing

String manipulation for text-based applications

Data Structures

Foundation for implementing complex data structures

Memory Representation Summary

Sequential Memory Storage

arr[0]	arr[1]	arr[2]	...

Contiguous memory addresses

- 1D arrays: stored in a single continuous block
- 2D arrays: stored in row-major order
- Strings: character arrays terminated with '\0'

Integration Example

```c
#include <stdio.h>
#include <string.h>

int main() {
    // Student records: 2D array
    int marks[3][3] = {
        {85, 76, 93},  // Student 1
        {80, 92, 78},  // Student 2
        {88, 82, 90}   // Student 3
    };

    // Student names: array of strings
    char names[3][20] = {"Alice", "Bob", "Charlie"};

    // Display student information
    for (int i = 0; i < 3; i++) {
        printf("%s: %d %d %d\n",
            names[i], marks[i][0],
            marks[i][1], marks[i][2]);
    }

    return 0;
}
```

Further Learning Resources

- C Programming Language (K&R)
- Online coding platforms
- Practice programming problems
- Q&A in lab sessions