Arrays in C++

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

1 Introduction to Arrays

In programming, an **array** is a collection of elements of the same data type stored in contiguous memory locations. Think of an array as a row of mailboxes in an apartment building - each mailbox is numbered (indexed), holds one item, and they're all lined up next to each other.

Imagine you need to store test scores for 50 students. Without arrays, you would need 50 separate variables: `score1`, `score2`, `score3`, and so on. This would be extremely tedious and impractical. Arrays solve this problem by allowing you to store all 50 scores in a single structure.

Arrays provide several key benefits:

- **Efficient Storage:** Store multiple values of the same type under one name
- **Easy Access:** Access elements using index numbers
- **Sequential Processing:** Perfect for loops and iterations
- **Memory Efficiency:** Elements stored contiguously in memory
- **Organization:** Group related data together logically

Key characteristics of arrays in C++:

1. **Fixed Size:** Array size must be determined at compile time or initialization
2. **Homogeneous:** All elements must be of the same data type
3. **Zero-indexed:** First element is at index 0, not 1
4. **Contiguous Memory:** Elements stored in consecutive memory locations

2 Array Declaration and Initialization

2.1 Basic Array Declaration

To declare an array, you specify the data type, array name, and size. The size must be a constant expression known at compile time.

Syntax:

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

Example: Basic array declarations

```
#include <iostream>
using namespace std;

int main() {
    // Declaring arrays of different types
    int scores[5];            // Array of 5 integers
    double temperatures[7];   // Array of 7 doubles
    char grades[10];          // Array of 10 characters
    bool flags[3];            // Array of 3 boolean values

    // Array size must be a constant
    const int SIZE = 8;
    int numbers[SIZE];        // Valid: SIZE is constant

    // The following would cause an error:
    // int n = 10;
    // int invalid[n];        // Error: n is not a constant expression

    cout << "Arrays declared successfully!" << endl;
    cout << "Size of scores array: " << sizeof(scores) / sizeof(
       scores[0]) << " elements" << endl;
    cout << "Size of temperatures array: " << sizeof(temperatures) /
       sizeof(temperatures[0]) << " elements" << endl;

    return 0;
}
```

Output:

```
Arrays declared successfully!
Size of scores array: 5 elements
Size of temperatures array: 7 elements
```

2.2 Array Initialization

Arrays can be initialized at the time of declaration in several ways:

Example: Different initialization methods

```
#include <iostream>
using namespace std;

int main() {
    // Method 1: Initialize with values
    int numbers[5] = {10, 20, 30, 40, 50};

    // Method 2: Partial initialization (rest filled with 0)
    int values[5] = {1, 2};  // values = {1, 2, 0, 0, 0}

    // Method 3: Initialize all elements to zero
    int zeros[5] = {0};       // All elements = 0
    int also_zeros[5] = {};  // Also all elements = 0

    // Method 4: Size determined by initializer list
    int auto_size[] = {5, 10, 15, 20};  // Size automatically becomes
        4

    // Method 5: Character array initialization
    char vowels[5] = {'a', 'e', 'i', 'o', 'u'};

    // Method 6: String as character array
    char name[] = "Alice";   // Automatically includes '\0' at end

    // Displaying initialized arrays
    cout << "numbers array: ";
    for (int i = 0; i < 5; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    cout << "values array (partial init): ";
    for (int i = 0; i < 5; i++) {
        cout << values[i] << " ";
    }
    cout << endl;

    cout << "auto_size array: ";
    for (int i = 0; i < 4; i++) {
        cout << auto_size[i] << " ";
    }
    cout << endl;

    cout << "vowels array: ";
    for (int i = 0; i < 5; i++) {
        cout << vowels[i] << " ";
    }
    cout << endl;

    cout << "name array: " << name << endl;

    return 0;
}
```

Output:

```
numbers array: 10 20 30 40 50
values array (partial init): 1 2 0 0 0
auto_size array: 5 10 15 20
vowels array: a e i o u
name array: Alice
```

3 Accessing Array Elements

Array elements are accessed using square brackets and an index number. Remember that array indices start at 0!

Example: Accessing and modifying array elements

```cpp
#include <iostream>
using namespace std;

int main() {
    int scores[5] = {85, 92, 78, 95, 88};

    // Accessing individual elements
    cout << "First score (index 0): " << scores[0] << endl;
    cout << "Second score (index 1): " << scores[1] << endl;
    cout << "Last score (index 4): " << scores[4] << endl;

    // Modifying elements
    cout << "\nModifying scores..." << endl;
    scores[2] = 82;  // Change third element from 78 to 82
    scores[0] = scores[0] + 5;  // Increase first score by 5

    cout << "Updated first score: " << scores[0] << endl;
    cout << "Updated third score: " << scores[2] << endl;

    // Using array elements in calculations
    int total = 0;
    for (int i = 0; i < 5; i++) {
        total = total + scores[i];
    }
    double average = total / 5.0;

    cout << "\nAll scores: ";
    for (int i = 0; i < 5; i++) {
        cout << scores[i] << " ";
    }
    cout << endl;
    cout << "Total: " << total << endl;
    cout << "Average: " << average << endl;

    return 0;
}
```

Output:

```
First score (index 0): 85
Second score (index 1): 92
Last score (index 4): 88

Modifying scores...
Updated first score: 90
Updated third score: 82

All scores: 90 92 82 95 88
Total: 447
Average: 89.4
```

3.1 Array Index Bounds

One of the most common errors in array programming is accessing an index that's out of bounds. C++ does not perform automatic bounds checking!

Example: Understanding array bounds

```cpp
#include <iostream>
using namespace std;

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

    cout << "Valid array accesses:" << endl;
    cout << "numbers[0] = " << numbers[0] << endl;  // Valid
    cout << "numbers[4] = " << numbers[4] << endl;  // Valid (last
        element)

    cout << "\nArray size: 5 elements" << endl;
    cout << "Valid indices: 0 to 4" << endl;

    // WARNING: The following are invalid but may not cause immediate
         errors
    // cout << "numbers[5] = " << numbers[5] << endl;   // INVALID:
        Out of bounds
    // cout << "numbers[-1] = " << numbers[-1] << endl; // INVALID:
        Negative index

    cout << "\nRemember: For an array of size N, valid indices are 0
        to N-1" << endl;

    // Safe way to iterate through array
    const int SIZE = 5;
    cout << "Safe iteration: ";
    for (int i = 0; i < SIZE; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Valid array accesses:
numbers[0] = 10
numbers[4] = 50

Array size: 5 elements
Valid indices: 0 to 4

Remember: For an array of size N, valid indices are 0 to N-1
Safe iteration: 10 20 30 40 50
```

4 Common Array Operations

4.1 Input and Output Operations

Example: Reading input into arrays

```cpp
#include <iostream>
using namespace std;

int main() {
    const int SIZE = 5;
    int numbers[SIZE];

    // Reading values into array
    cout << "Enter " << SIZE << " numbers:" << endl;
    for (int i = 0; i < SIZE; i++) {
        cout << "Number " << (i + 1) << ": ";
        cin >> numbers[i];
    }

    // Displaying array contents
    cout << "\nYou entered: ";
    for (int i = 0; i < SIZE; i++) {
        cout << numbers[i];
        if (i < SIZE - 1) {
            cout << ", ";
        }
    }
    cout << endl;

    // Reading characters
    char letters[3];
    cout << "\nEnter 3 letters: ";
    for (int i = 0; i < 3; i++) {
        cin >> letters[i];
    }

    cout << "Letters: ";
    for (int i = 0; i < 3; i++) {
        cout << letters[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Enter 5 numbers:
Number 1: 15
Number 2: 23
Number 3: 8
Number 4: 42
Number 5: 31

You entered: 15, 23, 8, 42, 31

Enter 3 letters: a b c
Letters: a b c
```

4.2 Finding Maximum and Minimum

Example: Finding max and min values

```cpp
#include <iostream>
using namespace std;

int main() {
    int numbers[] = {45, 23, 67, 12, 89, 34, 78};
    int size = 7;

    // Find maximum
    int max_value = numbers[0];  // Assume first element is max
    int max_index = 0;

    for (int i = 1; i < size; i++) {
        if (numbers[i] > max_value) {
            max_value = numbers[i];
            max_index = i;
        }
    }

    // Find minimum
    int min_value = numbers[0];  // Assume first element is min
    int min_index = 0;

    for (int i = 1; i < size; i++) {
        if (numbers[i] < min_value) {
            min_value = numbers[i];
            min_index = i;
        }
    }

    // Display results
    cout << "Array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    cout << "\nMaximum value: " << max_value << " at index " <<
       max_index << endl;
    cout << "Minimum value: " << min_value << " at index " <<
       min_index << endl;

    return 0;
}
```

Output:

```
Array: 45 23 67 12 89 34 78

Maximum value: 89 at index 4
Minimum value: 12 at index 3
```

4.3 Searching in Arrays

Example: Linear search

```
#include <iostream>
using namespace std;

int main() {
    int numbers[] = {15, 23, 8, 42, 31, 67, 19};
    int size = 7;
    int search_value;

    cout << "Array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    cout << "\nEnter a value to search: ";
    cin >> search_value;

    // Linear search
    bool found = false;
    int position = -1;

    for (int i = 0; i < size; i++) {
        if (numbers[i] == search_value) {
            found = true;
            position = i;
            break;  // Stop searching once found
        }
    }

    // Display result
    if (found) {
        cout << search_value << " found at index " << position <<
            endl;
    } else {
        cout << search_value << " not found in the array" << endl;
    }

    // Count occurrences
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (numbers[i] == search_value) {
            count++;
        }
    }
    cout << "Number of occurrences: " << count << endl;

    return 0;
}
```

Output:

```
Array: 15 23 8 42 31 67 19

Enter a value to search: 42
42 found at index 3
Number of occurrences: 1
```

4.4 Reversing an Array

Example: Reversing array elements

```
#include <iostream>
using namespace std;

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = 5;

    cout << "Original array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // Reverse the array by swapping elements
    for (int i = 0; i < size / 2; i++) {
        int temp = numbers[i];
        numbers[i] = numbers[size - 1 - i];
        numbers[size - 1 - i] = temp;
    }

    cout << "Reversed array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Original array: 10 20 30 40 50
Reversed array: 50 40 30 20 10
```

5 Arrays and Functions

Arrays can be passed to functions, allowing you to process array data in modular, reusable ways.

5.1 Passing Arrays to Functions

When you pass an array to a function, you're actually passing the address of the first element. This means the function can modify the original array.

Example: Basic array passing

```
#include <iostream>
using namespace std;

// Function to display array contents
void display_array(int arr[], int size) {
    cout << "Array: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Function to calculate sum
int calculate_sum(int arr[], int size) {
    int total = 0;
    for (int i = 0; i < size; i++) {
        total += arr[i];
    }
    return total;
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = 5;

    cout << "Original ";
    display_array(numbers, size);

    int sum = calculate_sum(numbers, size);
    cout << "Sum: " << sum << endl;

    return 0;
}
```

Output:

```
Original Array: 10 20 30 40 50
Sum: 150
```

5.2 Functions Returning Array Information

Example: Functions that analyze arrays

```
#include <iostream>
using namespace std;

// Find maximum value
int find_max(int arr[], int size) {
    int max_val = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}

// Find minimum value
int find_min(int arr[], int size) {
    int min_val = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }
    return min_val;
}


int main() {
    int data[] = {23, 45, 12, 67, 34, 89, 56};
    int size = 7;

    cout << "Array: ";
    for (int i = 0; i < size; i++) {
        cout << data[i] << " ";
    }
    cout << endl;

    cout << "Maximum: " << find_max(data, size) << endl;
    cout << "Minimum: " << find_min(data, size) << endl;

    return 0;
}
```

Output:

```
Array: 23 45 12 67 34 89 56
Maximum: 89
Minimum: 12
```

6 Multi-Dimensional Arrays

Multi-dimensional arrays are arrays of arrays. The most common is the two-dimensional array, which can be visualized as a table with rows and columns.

6.1 Two-Dimensional Arrays

Syntax:

```
data_type array_name[rows][columns];
```

Example: Declaring and initializing 2D arrays

```
#include <iostream>
using namespace std;

int main() {
    // Method 1: Initialize all elements
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Method 2: Partial initialization
    int partial[2][3] = {{1, 2}, {3}};  // Rest filled with 0

    // Method 3: Row-major order
    int sequential[2][3] = {1, 2, 3, 4, 5, 6};

    // Display the matrix
    cout << "3x4 Matrix:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "\nPartial initialization:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << partial[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
3x4 Matrix:
1 2 3 4
5 6 7 8
9 10 11 12

Partial initialization:
1 2 0
3 0 0
```

6.2 Working with 2D Arrays

Example: Common 2D array operations

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    const int ROWS = 3;
    const int COLS = 4;
    int scores[ROWS][COLS] = {
        {85, 90, 78, 92},
        {88, 76, 95, 89},
        {72, 84, 91, 87}
    };

    // Display the table
    cout << "Student Scores:" << endl;
    cout << "     Test1 Test2 Test3 Test4" << endl;
    for (int i = 0; i < ROWS; i++) {
        cout << "S" << (i + 1) << ":  ";
        for (int j = 0; j < COLS; j++) {
            cout << setw(5) << scores[i][j];
        }
        cout << endl;
    }

    // Calculate row averages (student averages)
    cout << "\nStudent Averages:" << endl;
    for (int i = 0; i < ROWS; i++) {
        int sum = 0;
        for (int j = 0; j < COLS; j++) {
            sum += scores[i][j];
        }
        double avg = static_cast<double>(sum) / COLS;
        cout << "Student " << (i + 1) << ": " << avg << endl;
    }

    // Calculate column averages (test averages)
    cout << "\nTest Averages:" << endl;
    for (int j = 0; j < COLS; j++) {
        int sum = 0;
        for (int i = 0; i < ROWS; i++) {
            sum += scores[i][j];
        }
        double avg = static_cast<double>(sum) / ROWS;
        cout << "Test " << (j + 1) << ": " << avg << endl;
    }

    return 0;
}
```

Output:

```
Student Scores:
    Test1 Test2 Test3 Test4
S1: 85 90 78 92
S2: 88 76 95 89
S3: 72 84 91 87

Student Averages:
Student 1: 86.25
Student 2: 87
Student 3: 83.5

Test Averages:
Test 1: 81.6667
Test 2: 83.3333
Test 3: 88
Test 4: 89.3333
```

6.3 Passing 2D Arrays to Functions

Example: Functions with 2D arrays

```
#include <iostream>
using namespace std;

const int ROWS = 3;
const int COLS = 3;

// Display 2D array
void display_matrix(int matrix[][COLS], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < COLS; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

// Calculate sum of all elements
int matrix_sum(int matrix[][COLS], int rows) {
    int total = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < COLS; j++) {
            total += matrix[i][j];
        }
    }
    return total;
}


int main() {
    int data[ROWS][COLS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    cout << "Original Matrix:" << endl;
    display_matrix(data, ROWS);

    cout << "\nSum of all elements: " << matrix_sum(data, ROWS) <<
       endl;


    return 0;
}
```

Output:

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

Sum of all elements: 45
5 found in matrix

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

7 Character Arrays and C-Strings

Character arrays have special significance in C++ as they can represent strings (sequences of characters).

7.1 C-String Basics

Example: Working with C-strings

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    // C-string declaration and initialization
    char name1[20] = "Alice";
    char name2[] = "Bob";
    char greeting[50];

    cout << "name1: " << name1 << endl;
    cout << "name2: " << name2 << endl;

    // String input
    cout << "\nEnter your name: ";
    cin.getline(greeting, 50);
    cout << "Hello, " << greeting << "!" << endl;

    // String length
    cout << "Length of name1: " << strlen(name1) << endl;

    // String copy
    char copy[20];
    strcpy(copy, name1);
    cout << "Copied string: " << copy << endl;

    // String concatenation
    char full_name[50] = "John ";
    strcat(full_name, "Smith");
    cout << "Full name: " << full_name << endl;

    // String comparison
    if (strcmp(name1, name2) == 0) {
        cout << "Names are equal" << endl;
    } else {
        cout << "Names are different" << endl;
    }

    return 0;
}
```

Output:

```
name1: Alice
name2: Bob

Enter your name: Charlie Brown
Hello, Charlie Brown!
Length of name1: 5
Copied string: Alice
Full name: John Smith
Names are different
```

8 Common Array Algorithms

8.1 Bubble Sort

Example: Bubble sort implementation

```
#include <iostream>
using namespace std;

void bubble_sort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void display(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int numbers[] = {64, 34, 25, 12, 22, 11, 90};
    int size = 7;

    cout << "Original array: ";
    display(numbers, size);

    bubble_sort(numbers, size);

    cout << "Sorted array: ";
    display(numbers, size);

    return 0;
}
```

Output:

```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

8.2 Selection Sort

Example: Selection sort implementation

```cpp
#include <iostream>
using namespace std;

void selection_sort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        // Swap minimum element with first element
        if (min_index != i) {
            int temp = arr[i];
            arr[i] = arr[min_index];
            arr[min_index] = temp;
        }
    }
}

void display(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int data[] = {29, 10, 14, 37, 13, 25, 18};
    int size = 7;

    cout << "Original array: ";
    display(data, size);

    selection_sort(data, size);

    cout << "Sorted array: ";
    display(data, size);

    return 0;
}
```

Output:

```
Original array: 29 10 14 37 13 25 18
Sorted array: 10 13 14 18 25 29 37
```

8.3 Binary Search

Example: Binary search in sorted array

```
#include <iostream>
using namespace std;

int binary_search(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;  // Found
        }
        else if (arr[mid] < target) {
            left = mid + 1;  // Search right half
        }
        else {
            right = mid - 1;  // Search left half
        }
    }

    return -1;  // Not found
}

int main() {
    int sorted_array[] = {11, 12, 22, 25, 34, 64, 90};
    int size = 7;
    int target;

    cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        cout << sorted_array[i] << " ";
    }
    cout << endl;

    cout << "\nEnter value to search: ";
    cin >> target;

    int result = binary_search(sorted_array, size, target);

    if (result != -1) {
        cout << target << " found at index " << result << endl;
    } else {
        cout << target << " not found in array" << endl;
    }

    return 0;
}
```

Output:

```
Sorted array: 11 12 22 25 34 64 90

Enter value to search: 25
25 found at index 3
```

9 Common Pitfalls and Best Practices

9.1 Common Pitfalls

- **Array Index Out of Bounds:** Accessing index outside valid range (0 to size-1)
- **Uninitialized Arrays:** Using array elements before assigning values
- **Array Size in Functions:** Arrays decay to pointers when passed to functions - must pass size separately
- **Confusing Array Size:** Remember array size is fixed at declaration
- **Off-by-One Errors:** Common in loop conditions (using $\leq$ instead of $<$)
- **Buffer Overflow:** Writing beyond array bounds can corrupt memory
- **Forgetting Null Terminator:** C-strings need `\0` at the end

9.2 Best Practices

- Always use constants for array sizes: `const int SIZE = 10;`
- Initialize arrays when possible to avoid garbage values
- Always pass array size as a parameter to functions
- Use meaningful variable names for indices (`i`, `j`, `row`, `col`)
- Check bounds before accessing array elements
- Use loops for repetitive array operations
- Comment complex array manipulations
- For modern C++, consider using `std::array` or `std::vector`
- Validate user input before using as array index
- Be careful with C-string functions to prevent buffer overflows

Example: Good practices demonstration

```
#include <iostream>
using namespace std;

const int MAX_SIZE = 100;  // Good: Use constant for size

// Good: Pass size explicitly
void safe_display(int arr[], int size) {
    // Good: Check for valid size
    if (size <= 0 || size > MAX_SIZE) {
        cout << "Invalid array size!" << endl;
        return;
    }

    // Good: Use < not <= to avoid off-by-one error
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    const int SIZE = 5;  // Good: Constant size
    int numbers[SIZE] = {0};  // Good: Initialize

    // Good: Get user input with validation
    cout << "Enter " << SIZE << " numbers:" << endl;
    for (int i = 0; i < SIZE; i++) {
        cout << "Number " << (i + 1) << ": ";
        cin >> numbers[i];
    }

    // Good: Pass both array and size
    safe_display(numbers, SIZE);

    return 0;
}
```

Output:

```
Enter 5 numbers:
Number 1: 10
Number 2: 20
Number 3: 30
Number 4: 40
Number 5: 50
10 20 30 40 50
```

10 Conclusion

Arrays are fundamental data structures in C++ that allow efficient storage and manipulation of multiple values of the same type. Understanding arrays is essential for effective programming and prepares you for more advanced data structures.

Key Points to Remember:

- Arrays store multiple elements of the same type in contiguous memory
- Array indices start at 0 and end at size-1
- Array size must be known at compile time (for static arrays)
- When passed to functions, arrays are passed by reference (address)
- C++ does not perform automatic bounds checking - programmer's responsibility
- Multi-dimensional arrays are arrays of arrays
- Character arrays can represent C-strings (null-terminated)
- Common operations include searching, sorting, finding max/min, and reversing
- Always validate array indices before access
- Use constants for array sizes and pass size to functions

Mastering arrays will prepare you for more advanced data structures like linked lists, stacks, queues, and dynamic arrays (vectors) in C++.