5. Arrays and Strings

1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

- Arrays in C++ are data structures used to store a collection of elements of the same data type in contiguous memory locations. They provide a way to store multiple values under a single variable name, accessed using an index.
- Key difference between Single-Dimensional & Multi-Dimensional:

Features	1D Array	2D/Mult-D Array
Dimensions	Single index arr[i]	Multi index arr[i][j] arr[i][j][k].
Structure	Simple Liner List	Matrix/table/higher-dimensional block.
Declaration Syntax	Type name[size]	Type name[sixe1][size2]
Initialization	{a, b, c}	{{a, b}, {c, d}}
Memory Layout	Contiguous	Contiguous, but split by rows (row-major).
Looping Logic	Single Loop	Nested Loop per dimension
Use Cases	Lists, static data sets.	Tables (spreadsheets), images, 3D data.

2. Explain string handling in C++ with examples.

- String handling in C++ primarily involves two approaches: C-style strings (character arrays) and the std::string class.

1. C-style Strings (Character Arrays):

C-style strings are null-terminated character arrays. They are essentially arrays of characters where the last character is a null character (\0) to signify the end of the string.

```
#include <iostream>
#include <cstring> // For C-style string functions

int main() {
    char greeting[20] = "Hello"; // Declaring and initializing a C-style string
    char name[10];
    std::cout << "Enter your name: ";
    std::cin >> name; // Input for C-style string (stops at whitespace)

    std::cout << "Greeting: " << greeting << std::endl;
    std::cout << "Name: " << name << std::endl;

    // Concatenation using strcat
    strcat(greeting, ", ");
    strcat(greeting, name);
    std::cout << "Combined: " << greeting << std::endl;

    // Length using strlen
    std::cout << "Length of combined string: " << strlen(greeting) << std::endl;

    return 0;
}
```

- ➢ Limitations:
  C-style strings require manual memory management and are prone to buffer overflows if not handled carefully. They lack many built-in functionalities for string manipulation compared to std::string.

2. std::string Class:

The std::string class, part of the <string> header, provides a more robust and convenient way to handle strings in C++. It manages memory dynamically and offers a rich set of member functions for various string operations.

```
#include <iostream>
#include <string>

int main() {
    std::string message = "Welcome"; // Declaring and initializing a std::string
    std::string user_input;

    std::cout << "Enter a message: ";
    std::getline(std::cin, user_input); // Input for std::string (reads entire line)

    std::cout << "Original message: " << message << std::endl;
    std::cout << "User input: " << user_input << std::endl;

    // Concatenation using operator+ or append()
    std::string combined = message + " to " + user_input;
    std::cout << "Combined string: " << combined << std::endl;

    // Length using length() or size()
    std::cout << "Length of combined string: " << combined.length() << std::endl;

    // Substring extraction
    std::string sub = combined.substr(0, 7); // Extracts "Welcome"
    std::cout << "Substring: " << sub << std::endl;

    // Finding a substring
    size_t found = combined.find("to");
    if (found != std::string::npos) {
        std::cout << "'to' found at index: " << found << std::endl;
    }

    return 0;
}
```

➢ Advantages of std::string:

i. **Dynamic Memory Management:** Handles memory allocation and deallocation automatically.

ii. **Rich Functionality:** Provides member functions for concatenation, comparison, searching, modification, and more.

iii. **Safety:** Reduces the risk of buffer overflows and other memory-related errors common with C-style strings.

iv. **Ease of Use:** Simplifies string manipulation tasks.

3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

- Arrays in C++ can be initialized at the time of declaration using an initializer list enclosed in curly braces {}.
  - **1D Array Initialization:**

    A one-dimensional array can be initialized by providing a comma-separated list of values.

    ```
    // Explicitly specifying size
    int numbers[5] = {10, 20, 30, 40, 50};

    // Implicitly determining size from initializer list
    int grades[] = {85, 92, 78, 95};

    // Partial initialization (remaining elements are zero-initialized)
    int scores[5] = {100, 90}; // scores will be {100, 90, 0, 0, 0}

    // Zero-initializing all elements
    int zeros[3] = {}; // zeros will be {0, 0, 0}
    ```

  - **2D Array Initialization**

    A two-dimensional array can be initialized using nested initializer lists, where each inner list represents a row.

    ```
    // Explicitly specifying dimensions
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

    // Implicitly determining row size, column size must be specified
    int grid[][2] = {{10, 20}, {30, 40}, {50, 60}};

    // Initializing sequentially without nested braces (values fill row by row)
    int sequential_matrix[2][2] = {1, 2, 3, 4}; // Equivalent to {{1, 2}, {3, 4}}
    ```

// Partial initialization (remaining elements are zero-initialized)
int partial_matrix[2][2] = {{1}}; // partial_matrix will be {{1, 0}, {0, 0}}

4. Explain string operations and functions in C++.

- C++ offers robust string manipulation capabilities primarily through the std::string class, part of the C++ Standard Library, and also supports C-style character arrays.

❖ **std::string Operations and Functions:**

➢ **Declaration and Initialization.**

```
std::string s1 = "Hello";
std::string s2("World");
std::string s3; // Empty string
```

i. **Concatenation:**
Using the + operator: std::string combined = s1 + ", " + s2;
Using append(): s1.append(", World!");

ii. **Length and Size:**
length() or size(): Returns the number of characters in the string.

```
std::string text = "Example";
int len = text.length(); // len will be 7
```

➢ **Accessing Characters:**

i. Using array-like indexing: char firstChar = text[0];
ii. Using at(): char secondChar = text.at(1); (provides bounds checking)

➢ **Comparison:**

i. Using comparison operators (==, !=, <, >, etc.): if (s1 == s2) { ... }
ii. Using compare(): Returns 0 for equality, a negative value if the calling string is lexicographically smaller, and a positive value if larger.

➢ **Substrings:**

i. substr(pos, len): Extracts a substring starting at pos with len characters.

```
std::string original = "Programming";
std::string sub = original.substr(3, 4); // sub will be "gram"
```

- **Searching:**
    1. find(substring): Returns the index of the first occurrence of substring, or std::string::npos if not found.

- **Modification:**
    1. replace(pos, len, new_string): Replaces a portion of the string.
    2. insert(pos, new_string): Inserts new_string at pos.
    3. erase(pos, len): Erases characters from pos for len.
    4. push_back(char): Adds a character to the end.
    5. pop_back(): Removes the last character.

- **Input/Output:**
    1. std::cin >> str; (reads until whitespace)
    2. std::getline(std::cin, str); (reads an entire line, including spaces)
    3. std::cout << str;

- **C-style String Functions (using char arrays and <cstring>):**

- While `std::string` is generally preferred in modern C++, C-style strings and their associated functions are still available:
    1. strlen(char_array): Returns the length of a null-terminated C-style string.
    2. strcpy(dest, src): Copies src to dest.
    3. strcat(dest, src): Concatenates src to dest.
    4. strcmp(str1, str2): Compares two C-style strings lexicographically.

- std::string is generally safer and more convenient due to automatic memory management, operator overloading, and a rich set of member functions. C-style strings require manual memory management and are prone to buffer overflows if not handled carefully. Use std::string for most modern C++ applications.