Recursion in C++

Recursion is a process where a function **calls itself** directly or indirectly until a base condition is met.

- A recursive function must have a **base case** (to stop recursion).
- Without a base case, it leads to **infinite recursion** (stack overflow).

Types of Recursion

1. Direct Recursion

A function calls **itself** directly.

Example: Factorial

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

int factorial(int n) {
    if (n == 0 || n == 1)   // base case
        return 1;
    else
        return n * factorial(n - 1);  // recursive call
}

int main() {
    cout << "Factorial of 5 = " << factorial(5);
    return 0;
}
```

Output:

```
Factorial of 5 = 120
```

2. Indirect Recursion

A function calls **another function**, which in turn calls the first function.

Example:

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

void funcA(int n);
void funcB(int n);

void funcA(int n) {
    if (n > 0) {
        cout << n << " ";
        funcB(n - 1);  // call to funcB
    }
}
```

```
void funcB(int n) {
    if (n > 1) {
        cout << n << " ";
        funcA(n / 2);  // call back to funcA
    }
}

int main() {
    funcA(10);
    return 0;
}
```

Examples of Recursion

1. Fibonacci Series

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

int fibonacci(int n) {
    if (n == 0) return 0;   // base case
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);  // recursive call
}

int main() {
    cout << "Fibonacci(6) = " << fibonacci(6);
    return 0;
}
```

Output:

```
Fibonacci(6) = 8
```

2. Sum of Natural Numbers

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

int sum(int n) {
    if (n == 0) return 0;       // base case
    return n + sum(n - 1);      // recursive call
}

int main() {
    cout << "Sum of first 5 numbers = " << sum(5);
    return 0;
}
```

Output:

```
Sum of first 5 numbers = 15
```

Advantages of Recursion

- Makes code **simpler and shorter**.

- Useful for problems like factorial, Fibonacci, Tower of Hanoi, tree/graph traversals.

Disadvantages of Recursion

- More memory usage (due to function call stack).
- Slower than loops in many cases.
- May cause **stack overflow** if base case is missing or wrong.

Recursion vs Iteration

Aspect	Recursion	Iteration (Loop)
Definition	Function calls itself	Repeats code using loops (`for`, `while`)
Memory Usage	More (stack frames created)	Less
Speed	Slower (overhead of calls)	Faster
Readability	More readable for mathematical problems	More suitable for simple repetitive tasks
Risk	Stack overflow (if base case missing)	Infinite loop (if condition missing)

One-Dimensional Array

A 1D array is a collection of elements of the same type stored **contiguously** in memory.

Initialization

```
int arr[5];              // uninitialized
int arr2[5] = {1,2,3,4,5}; // initialized
int arr3[] = {10,20,30};   // size inferred
```

Traversal

```
for (int i = 0; i < 5; i++) {
    cout << arr2[i] << " ";
}
```

Basic Operations

- **Access element:** `arr[i]`
- **Update element:** `arr[2] = 10;`
- **Sum of elements:**

```
int sum = 0;
for(int i=0; i<5; i++)
    sum += arr2[i];
```

Input and Print Elements of a 1D Array

Step 1: Declare the array

Ask the user for the size of the array then declare an array of that size.

```
int n;
cin >> n;         // size of array
int arr[n];       // declare array
```

Step 2: Take input from the user

Use a `for` loop to input elements.

```
for (int i = 0; i < n; i++) {
    cin >> arr[i];   // input element at index i
}
```

Step 3: Print elements of the array

Use a `for` loop to display the elements.

```
for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";   // print element at index i
}
cout << endl;
```

Two-Dimensional Array

A 2D array is like a **matrix**, elements are stored in rows and columns.

Initialization

```
int arr[2][3];                   // uninitialized
int arr2[2][3] = {{1,2,3},{4,5,6}}; // initialized
int arr3[][3] = {{1,2,3},{4,5,6}};   // size inferred for rows
```

Traversal

```
for(int i = 0; i < 2; i++) {
    for(int j = 0; j < 3; j++) {
        cout << arr2[i][j] << " ";
    }
    cout << endl;
}
```

Basic Operations

- **Access element:** `arr[i][j]`
- **Update element:** `arr[1][2] = 10;`
- **Sum of all elements:**

```
int sum = 0;
for(int i=0;i<2;i++){
    for(int j=0;j<3;j++){
        sum += arr2[i][j];
    }
}
```

Transpose of a 2D Array

- The **transpose** of a matrix is obtained by **swapping rows with columns**.
- If original array is `arr[rows][cols]`, the transpose is `arrT[cols][rows]`.

Algorithm

1. Create a new array `arrT[cols][rows]`.
2. For each element `arr[i][j]`, assign `arrT[j][i] = arr[i][j]`.
3. Print the transpose array.

Example Program: Transpose

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

int main() {
    int rows, cols;
    cout << "Enter number of rows and columns: ";
    cin >> rows >> cols;

    int arr[rows][cols];

    // Input array elements
    cout << "Enter elements of array:\n";
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            cin >> arr[i][j];
        }
    }

    // Create transpose array
    int arrT[cols][rows];
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            arrT[j][i] = arr[i][j];
        }
    }

    // Print transpose
    cout << "Transpose of the array:\n";
    for(int i = 0; i < cols; i++) {
        for(int j = 0; j < rows; j++) {
            cout << arrT[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

String Class vs Character Arrays in C++

1. Character Arrays

- A **character array** is a sequence of characters stored in contiguous memory.

- Must end with a **null character** `'\0'` to indicate the end of the string.

Declaration & Initialization

```
char str1[20];              // uninitialized
char str2[20] = "Hello";    // initialized
char str3[] = "World";      // size inferred
```

Traversal

```
for (int i = 0; str2[i] != '\0'; i++) {
    cout << str2[i] << " ";
}
```

Notes:

- Fixed size unless dynamically allocated with `new`.
- Manual handling required (`'\0'`, memory management).

Functions on Character Arrays (<cstring>)

Function	Description	Example
`strlen(str)`	Returns length of string (excluding `\0`)	`int len = strlen(str);`
`strcpy(dest, src)`	Copies `src` into `dest`	`strcpy(s2, s1);`
`strcat(dest, src)`	Concatenates `src` to end of `dest`	`strcat(s1, s2);`
`strcmp(s1, s2)`	Compares two strings (0 if equal, <0 if s1<s2, >0 if s1>s2)	`int res = strcmp(s1, s2);`
`strncpy(dest, src, n)`	Copies first `n` characters of `src` to `dest`	`strncpy(s2, s1, 3);`
`strchr(str, ch)`	Returns pointer to first occurrence of character `ch`	`char* p = strchr(str, 'e');`
`strstr(str, substr)`	Returns pointer to first occurrence of substring	`char* p = strstr(str, "lo");`

Example:

```
char str1[20] = "Hello";
char str2[20] = "World";

cout << "Length of str1: " << strlen(str1) << endl;
strcat(str1, str2);
cout << "After concatenation: " << str1 << endl;
```

2. String Class (std::string)

- `string` is a C++ class that represents a sequence of characters.
- Memory management is automatic.

Declaration & Initialization

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

string str1 = "Hello";
string str2("World");
```

Methods

Method	Description	Example
`length()` or `size()`	Returns number of characters	`int n = str1.length();`
`append()`	Adds another string at the end	`str1.append(str2);`
`+` operator	Concatenates two strings	`string str3 = str1 + str2;`
`at(index)`	Access character at given index	`char c = str1.at(1);`
`empty()`	Returns true if string is empty	`bool b = str1.empty();`
`substr(pos, len)`	Returns substring from `pos` of length `len`	`string s = str1.substr(0, 3);`
`find(str)`	Returns index of first occurrence of substring	`int idx = str1.find("lo");`
`replace(pos, len, str)`	Replace part of string from `pos` for `len` chars	`str1.replace(0, 2, "Hi");`
`erase(pos, len)`	Erases part of string	`str1.erase(0, 2);`
`c_str()`	Converts string to `const char*`	`const char* s = str1.c_str();`

Example:

```
string str1 = "Hello";
string str2 = "World";

cout << "Length of str1: " << str1.length() << endl;
string str3 = str1 + str2;
cout << "Concatenated string: " << str3 << endl;

str3.replace(0, 5, "Hi");
cout << "After replace: " << str3 << endl;
```

Key Differences

Feature	Character Array (`char[]`)	String Class (`std::string`)
Memory	Fixed-size or manual dynamic	Dynamic, automatic
Null-Terminator	Must add manually	Automatic
Concatenation	`strcat()`	`+` or `append()`
Comparison	`strcmp()`	`==`, `!=`, `<`, `>`
Safety	Less safe, prone to overflow	Safer, exceptions handled

Feature	Character Array (`char[]`)	String Class (`std::string`)
Ease of Use	Low-level, manual	High-level, easier to manipulate

Notes:

- Use **character arrays** only for C-style string operations or low-level tasks.
- Prefer **string class** in modern C++ for safety and convenience.
- Character array functions are from `<cstring>`; string class methods are built into the `std::string` class.