# Stuctured Programming Methodology

Module 4:

Functions

## Functions

- User Defined Functions: Need, Function Declaration and Definition, Return Values, Function Calls, Passing Arguments to a Function by Value, Recursive functions, String Handling functions (in-built)

## Functions

- A function is a block of statements that performs a specific task. Let’s say you are writing a C program and you need to perform a same task in that program more than once.
- In such case you have two options:

  a) Use the same set of statements every time you want to perform the task

  b) Create a function to perform that task, and just call it every time you need to perform that task.

- Using option (b) is a good practice and a good programmer always uses functions while writing code in C/C++.
- C++ also supports advanced features like function overloading, default arguments, and inline functions, which give more flexibility compared to C.

## Functions

**Why we need functions in C++**

Functions are used because of following reasons –

- To improve the **readability of code**.
- Improves the **reusability** of the code, same function can be used in any program rather than writing the same code from scratch.
- **Debugging** of the code **would be easier** if you use functions, as errors are easy to be traced.
- **Reduces the size of the code**, duplicate set of statements are replaced by function calls.

## Functions

**Types of functions**

**1) Predefined standard library functions**

- Standard library functions are also known as built-in functions.
- Functions such as  cout, sqrt(), abs(), and getline() etc are standard library functions.
- These functions are already defined in header files (files with .h extensions are called header files such as <iostream>, <cmath>, or <string>.), so we just call them whenever there is a need to use them.

**Note:**

main() is a special user-defined function that must be present in every C++ program.
It is the entry point of the program where execution begins. The compiler doesn't provide it for you, you write it yourself.

## Functions

**Types of functions**

**2) User Defined functions**

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

Now we will learn how to create user defined functions and how to use them in C++

## Functions
## User Defined functions : Syntax of a function

```
return_type function_name (argument list)
{
    Set of statements – Block of code
}
```

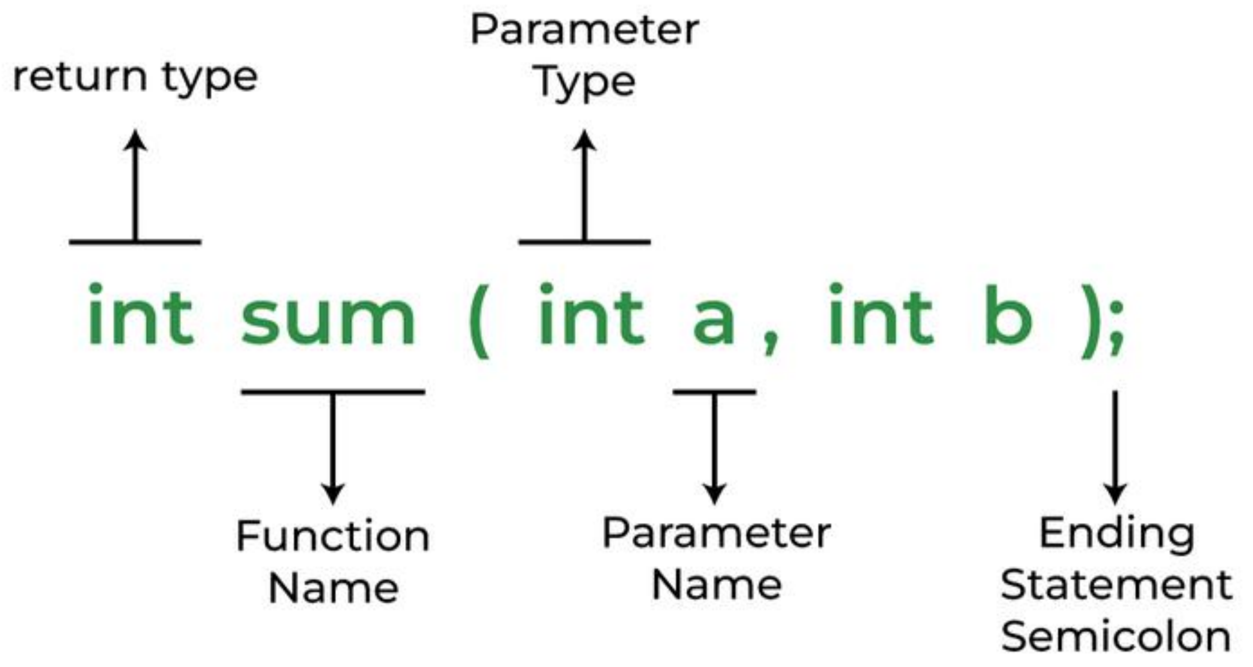
**return_type:** Return type can be of any data type such as int, double, char, void, short etc.

**function_name:** It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.

**argument list:** Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

**Block of code:** Set of C statements, which will be executed whenever a call will be made to the function.

## Example

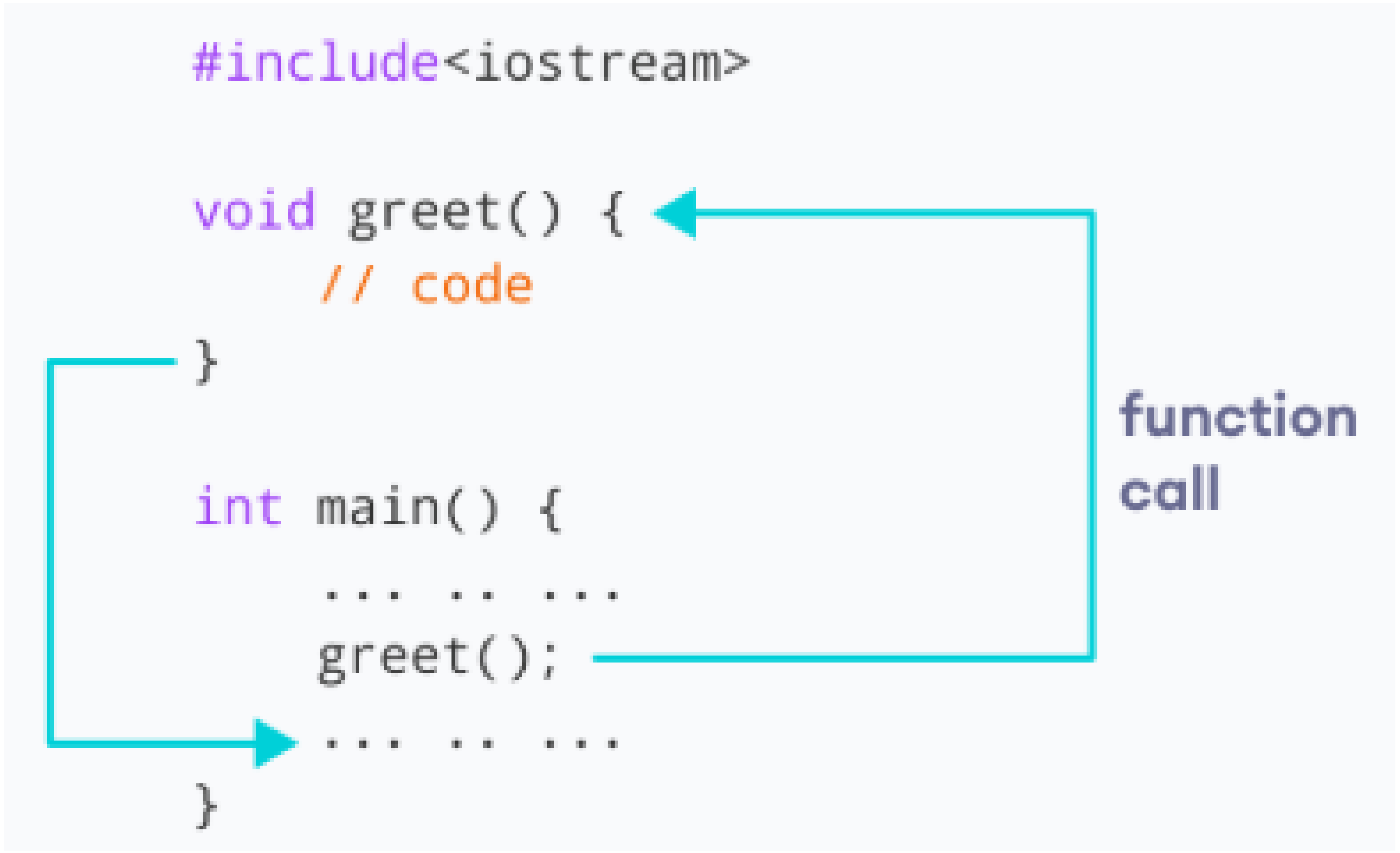

return type
Parameter Type
int sum ( int a , int b );
Function Name
Parameter Name
Ending Statement Semicolon

# Function Declaration and Definition

- A function consist of 3 parts:
- **Declaration:**
- the function's name, return type, and parameters (if any)
- **Definition:**
- the body of the function (code to be executed)
- **Function Calls**:
- A function call is a statement that instructs the compiler to execute the function.

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition

 void myFunction() {
   cout << "I just got executed!";
 }
```

# How function works in c++



```
#include<iostream>

void greet() {
    // code
}

int main() {
    ... .. ...
    greet();
    ... .. ...
}
```

## Functions

Lets take an example – **Suppose you want to create a function to add two integer variables.**

For example lets take the name addition for this function.

return_type addition(argument list)

This function addition adds two integer variables, which means we need two integer variable as input, lets provide two integer parameters in the function signature. The function signature would be –

return_type addition(int num1, int num2)

The result of the sum of two integers would be integer only. Hence function should return an integer value – we got our **return type** – It would be integer –

int addition(int num1, int num2);

So we got our function prototype or signature. Now we can implement the logic in C program like this:

## Example

- double AreaR(double,double);// function declaration
- double AreaR(double x,double y);//function definition
- AreaR(a,b);//function calling

```
#include <iostream>
using namespace std;
double AreaR(double length, double width) {
    return length * width;
}
int main() {
    double rectLength, rectWidth;
    cout << "Enter the length of the rectangle: ";  // Prompt the user for input
    cin >> rectLength;
   cout << "Enter the width of the rectangle: ";
   cin >> rectWidth;
       double area = AreaR(rectLength, rectWidth); // Call the function to calculate the area
     cout << "The area of the rectangle is: " << area << endl;   // Display the result
    return 0;
}
```

## Functions

## Creating a void user defined function that doesn’t return anything

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

// Function with return type void and no parameters
void introduction() {
    cout << "Hi" << endl;
    cout << "My name is Programming" << endl;
    cout << "How are you?" << endl;
    // No return statement since return type is void
}

int main() {
    // Calling function
    introduction();
    return 0;
}
```

## Functions

**Few Points to Note regarding functions in C:**

1) main() in C program is also a function.

2) Each C program must have at least one function, which is main().

3) There is no limit on number of functions; A C program can have any number of functions.

4) A function can call itself and it is known as "Recursion".

## Functions

## C Functions Terminologies that must remembered

**return type:** Data type of returned value. It can be void also, in such case function doesn’t return any value.

Note: for example, if function return type is char, then function should return a value of char type and while calling this function the main() function should have a variable of char data type to store the returned value.

## Functions

## Task

Write function for

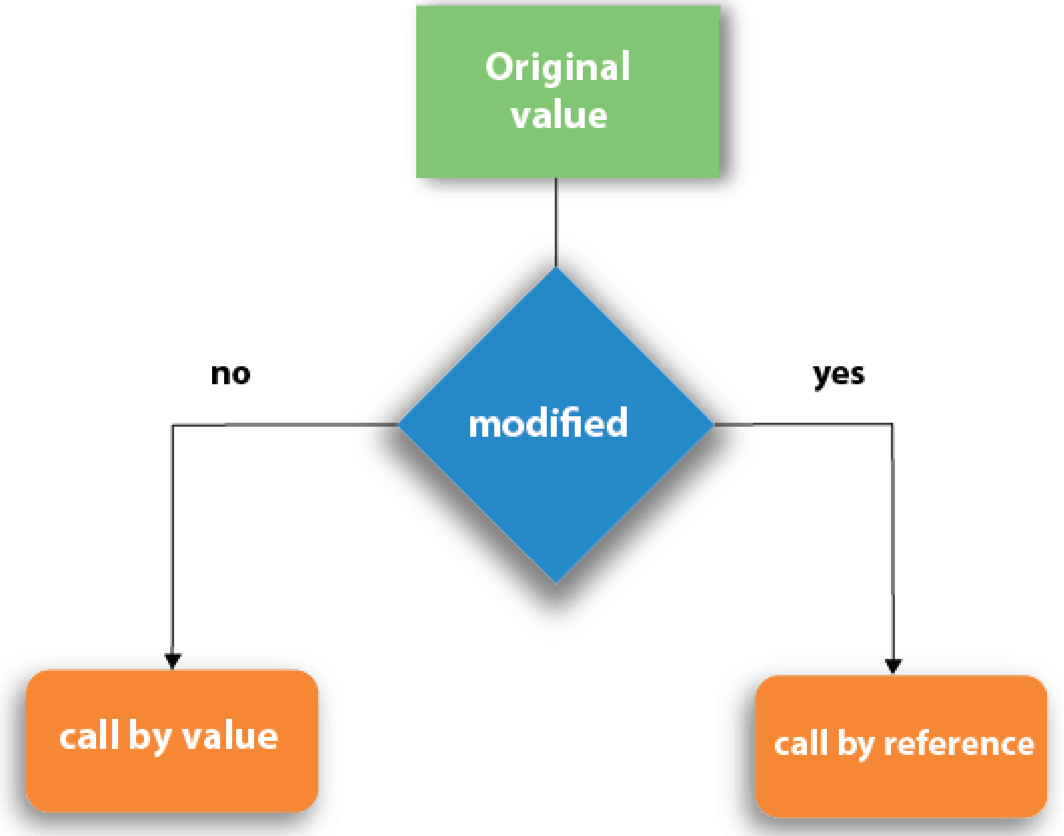
- addition of two integers,
- multiplication of two float numbers and
- display of character

## Functions

## Function Calls, Passing Arguments to a Function by Value

**1) Function – Call by value method** – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

**2) Function – Call by reference method** – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

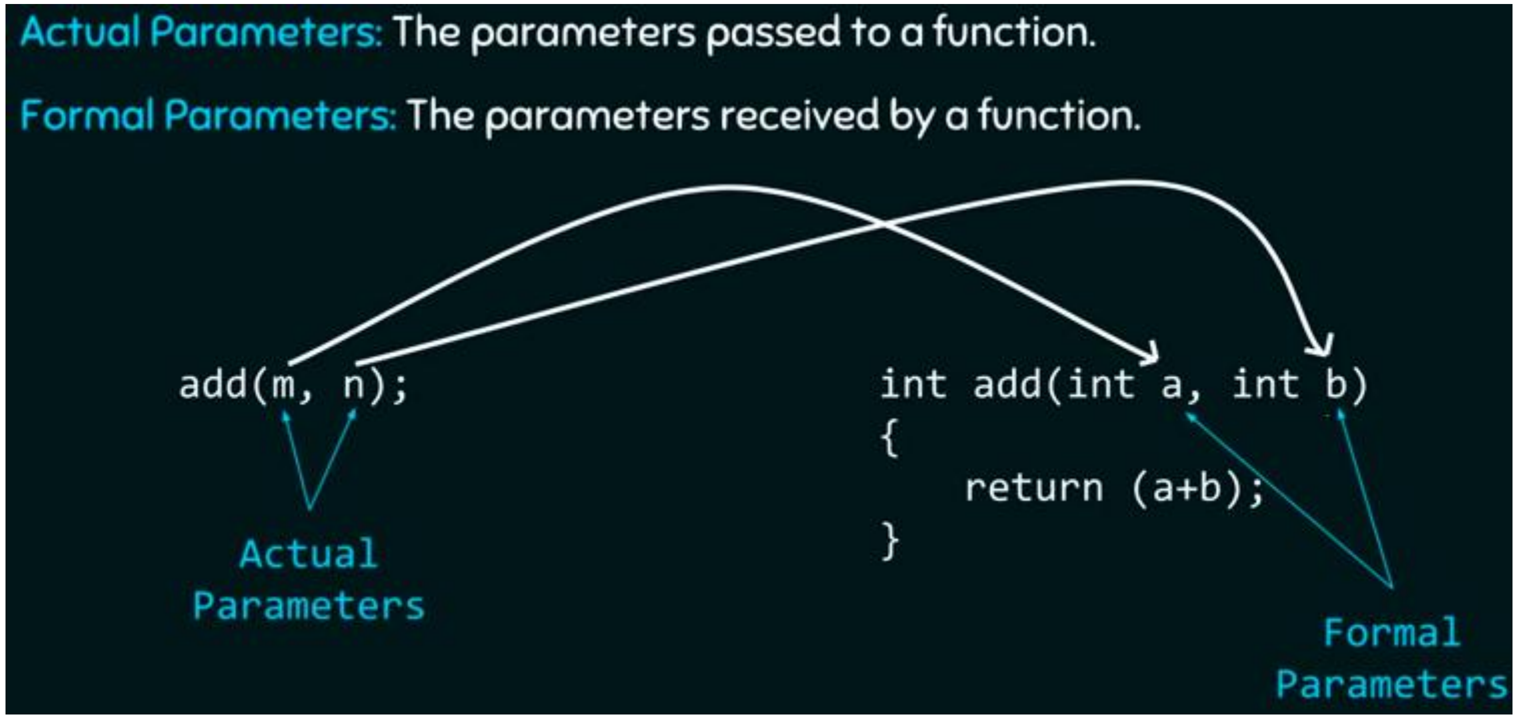

Original value
modified
no
yes
call by value
call by reference

## Functions

### Function – Call by value method

Actual parameters: The parameters that appear in function calls.
(eg.var1, var2) (Arguments)

Formal parameters: The parameters that appear in function declarations.
(num1, num2) (parameters)


Actual Parameters: The parameters passed to a function.

Formal Parameters: The parameters received by a function.

```
add(m, n);
```

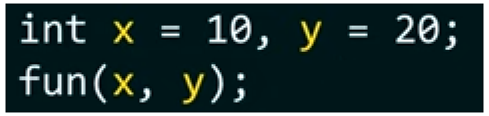
```
int add(int a, int b)
{
    return (a+b);
}
```

## Functions

### What is Function Call By value?

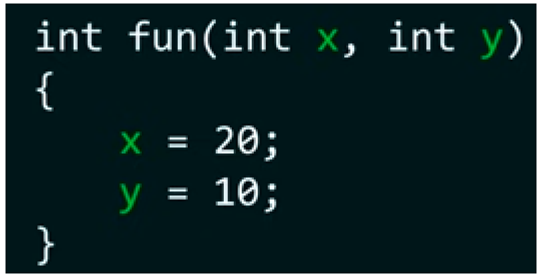
- When we pass the actual parameters while calling a function then this is known as function call by value.
- In this case the values of actual parameters are copied to the formal parameters.
- Thus operations performed on the formal parameters don’t reflect in the actual parameters.
- Example:

Function calling

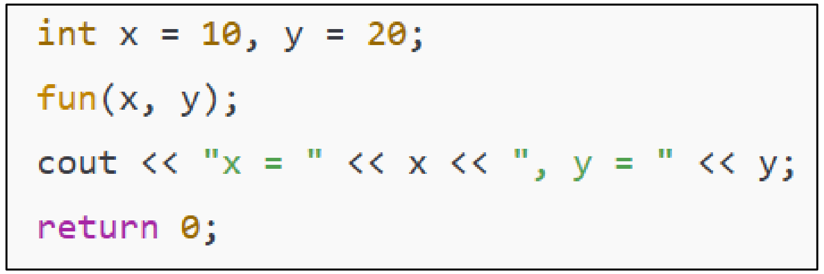


```
int x = 10, y = 20;
fun(x, y);
```

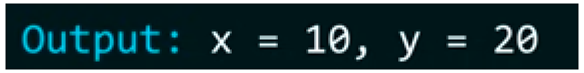
Function definition



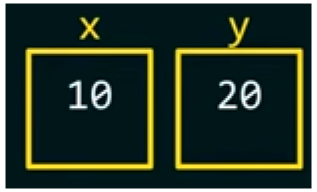
```
int fun(int x, int y)
{
    x = 20;
    y = 10;
}
```

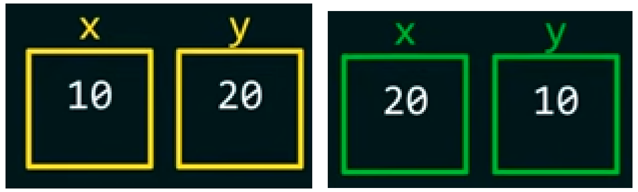


```
int x = 10, y = 20;
fun(x, y);
cout << "x = " << x << ", y = " << y;
return 0;
```

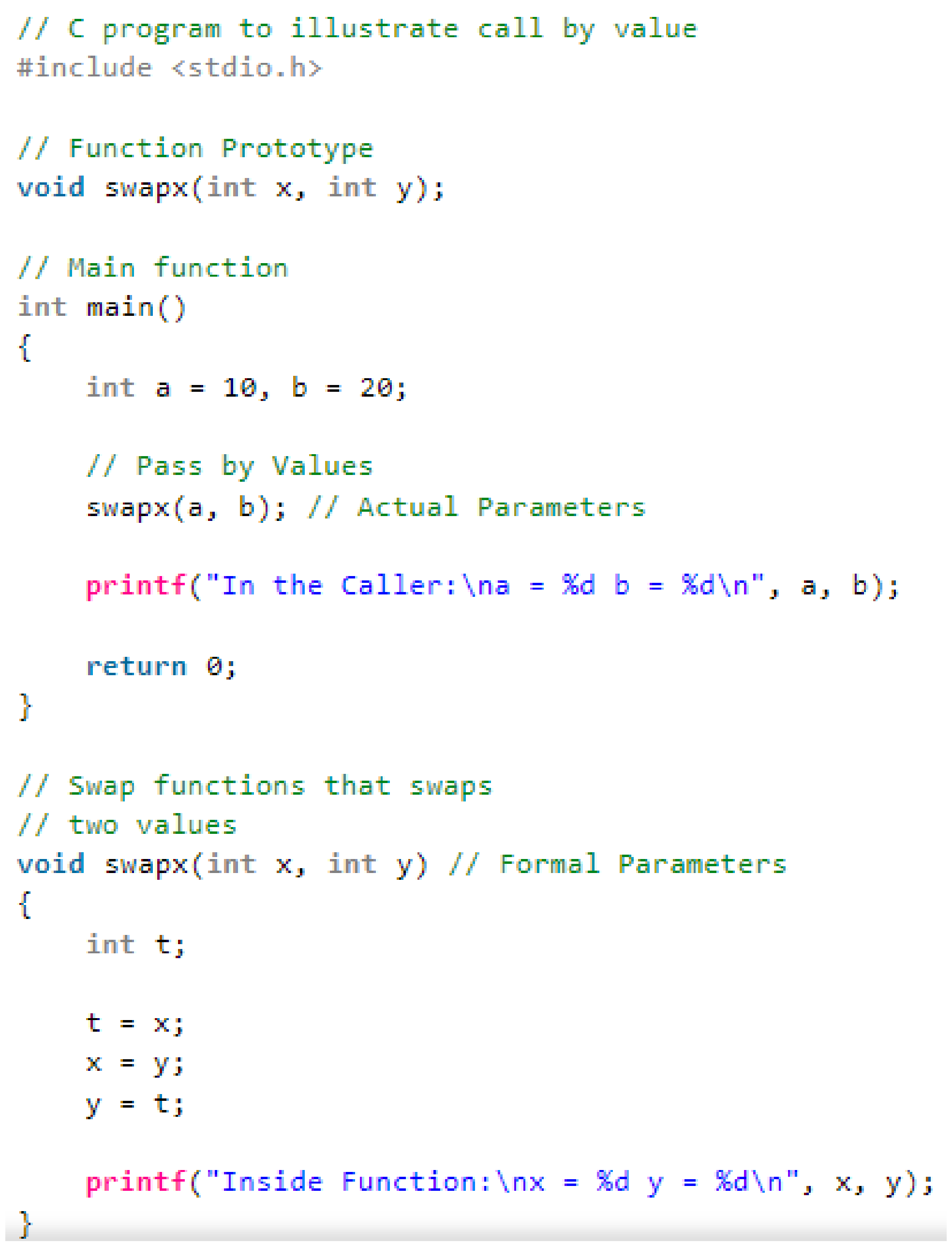


```
Output: x = 10, y = 20
```





Example:



```c
// C program to illustrate call by value
#include <stdio.h>

// Function Prototype
void swapx(int x, int y);

// Main function
int main()
{
    int a = 10, b = 20;

    // Pass by Values
    swapx(a, b); // Actual Parameters

    printf("In the Caller:\na = %d b = %d\n", a, b);

    return 0;
}

// Swap functions that swaps
// two values
void swapx(int x, int y) // Formal Parameters
{
    int t;

    t = x;
    x = y;
    y = t;

    printf("Inside Function:\nx = %d y = %d\n", x, y);
}
```

# Functions

## Function – Call by value method

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

int increment(int var) {
    var = var + 1;
    return var;
}

int main() {
    int num1 = 20;
    int num2 = increment(num1);

    cout << "num1  value is: " << num1 << endl;
    cout << "num2 value is: " << num2 << endl;

    return 0;
}
```

Output: ?

## Functions

### Function – Call by value method

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

int increment(int var) {
    var = var + 1;
    return var;
}

int main() {
    int num1 = 20;
    int num2 = increment(num1);

    cout << "num1 value is: " << num1 << endl;
    cout << "num2 value is: " << num2 << endl;

    return 0;
}
```

Output:

```
num1 value is: 20
num2 value is: 21
```

## Functions

**Swapping numbers using Function Call by Value**

```
#include <iostream>
using namespace std;
void swapnum(int var1, int var2) {
    int tempnum;
    tempnum = var1;
    var1 = var2;
    var2 = tempnum;

    // Changes happen only inside this function (local copy),
    // not reflected in main() because it's call by value.
}
int main() {
    int num1 = 35, num2 = 45;
    cout << "Before swapping: " << num1 << ", " << num2 << endl;
    // calling swap function
    swapnum(num1, num2);
    cout << "After swapping: " << num1 << ", " << num2 << endl;
    return 0;
}
```

Output:?

## Functions

**Swapping numbers using Function Call by Value**

```
#include <iostream>
using namespace std;
void swapnum(int var1, int var2) {
    int tempnum;
    tempnum = var1;
    var1 = var2;
    var2 = tempnum;

    // Changes happen only inside this function (local copy),
    // not reflected in main() because it's call by value.
}
int main() {
    int num1 = 35, num2 = 45;
    cout << "Before swapping: " << num1 << ", " << num2 << endl;
    // calling swap function
    swapnum(num1, num2);
    cout << "After swapping: " << num1 << ", " << num2 << endl;
    return 0;
}
```

Output:

```
Before swapping: 35, 45
After swapping: 35, 45
```

The reason is – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

# Functions

## Function – Call by reference method

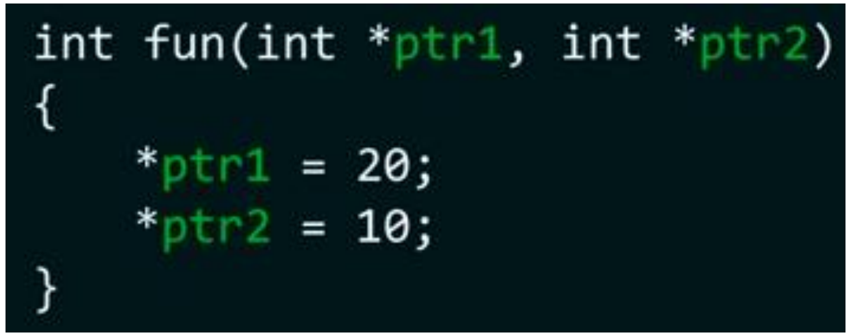
Actual parameters: The parameters that appear in function calls.
(eg.var1, var2)

Formal parameters: The parameters that appear in function declarations.
(num1, num2)

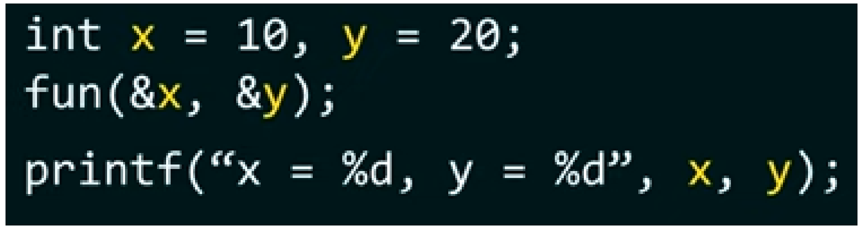
What is Function Call By reference?

Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

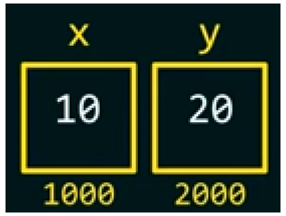
## Example:

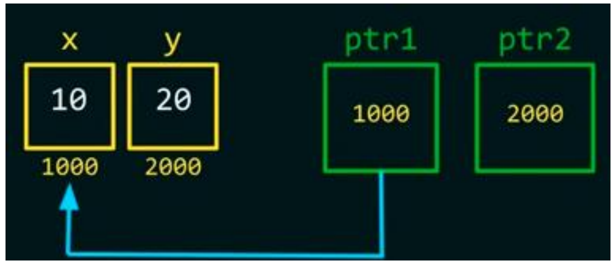


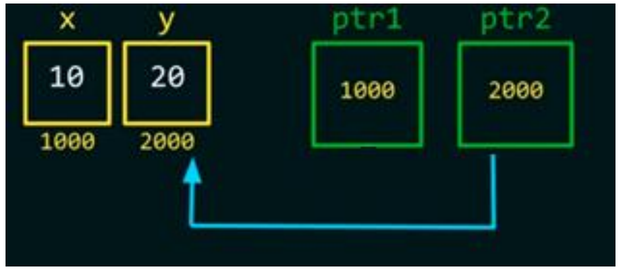
```
int fun(int *ptr1, int *ptr2)
{
    *ptr1 = 20;
    *ptr2 = 10;
}
```

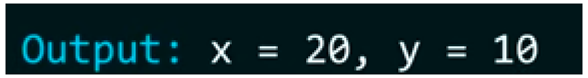


```
int x = 10, y = 20;
fun(&x, &y);
printf("x = %d, y = %d", x, y);
```

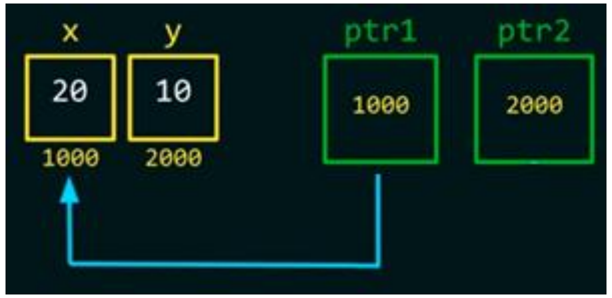


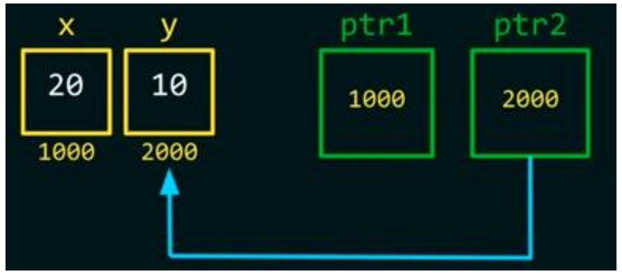


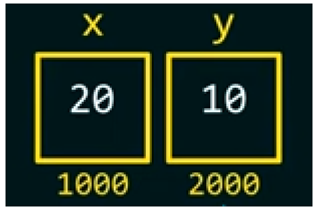




```
Output: x = 20, y = 10
```







# Functions

## Swapping numbers using Function Call by reference

```
#include <iostream>
using namespace std;
// Call by reference using reference variables
void swapnum(int &var1, int &var2) {
    int tempnum;
    tempnum = var1;
    var1 = var2;
    var2 = tempnum;
}
int main() {
    int num1 = 35, num2 = 45;
    cout << "Before swapping:";
    cout << "\nnum1  value is " << num1;
    cout << "\nnum2 value is " << num2 << endl;
    swapnum(num1, num2);                    // calling swap function
    cout << "\nAfter swapping:";
    cout << "\nnum1  value is " << num1;
    cout << "\nnum2 value is " << num2 << endl;
    return 0;
}
```

Output:?

# Functions

**Swapping numbers using Function Call by reference**

```
#include <iostream>
using namespace std;
// Call by reference using reference variables
void swapnum(int &var1, int &var2) {
    int tempnum;
    tempnum = var1;
    var1 = var2;
    var2 = tempnum;
}
int main() {
    int num1 = 35, num2 = 45;
    cout << "Before swapping:";
    cout << "\nnum1 value is " << num1;
    cout << "\nnum2 value is " << num2 << endl;
    swapnum(num1, num2);                // calling swap function
    cout << "\nAfter swapping:";
    cout << "\nnum1 value is " << num1;
    cout << "\nnum2 value is " << num2 << endl;
    return 0;
}
```

Output:

```
Before swapping:
num1 value is 35
num2 value is 45
After swapping:
num1 value is 45
num2 value is 35
```

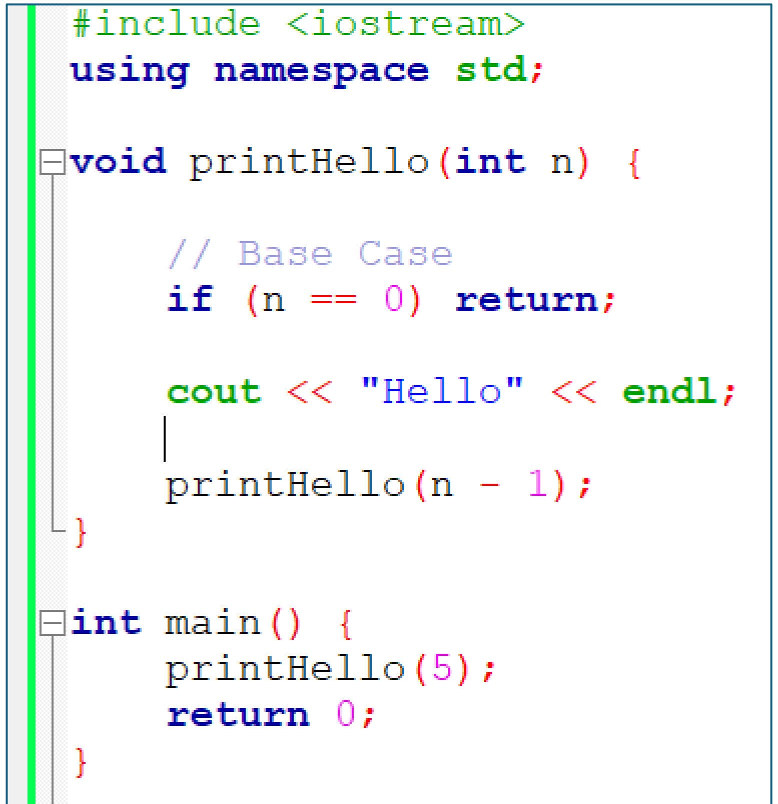
The values of the variables have been changed after calling the swapnum() function because the swap happened on the addresses of the variables num1 and num2.

## Exercise

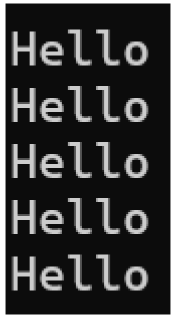
- Write a program to check whether given number is pallindrome or not. (Read number as user input).
- Expected op:
  - Enter the number: 123
  - Given number is not palindrome
  - Enter the number: 121
  - Given number is palindrome

# Recursion

- **Recursion** is a programming technique where a **function calls itself** directly or indirectly to solve a problem.
- Each recursive call **reduces the problem** into a smaller subproblem until it reaches a **base case**, which stops the recursion.



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

void printHello(int n) {

    // Base Case
    if (n == 0) return;

    cout << "Hello" << endl;

    printHello(n - 1);
}

int main() {
    printHello(5);
    return 0;
}
```



```
Hello
Hello
Hello
Hello
Hello
```

# Why do we use recursion? (few reasons)

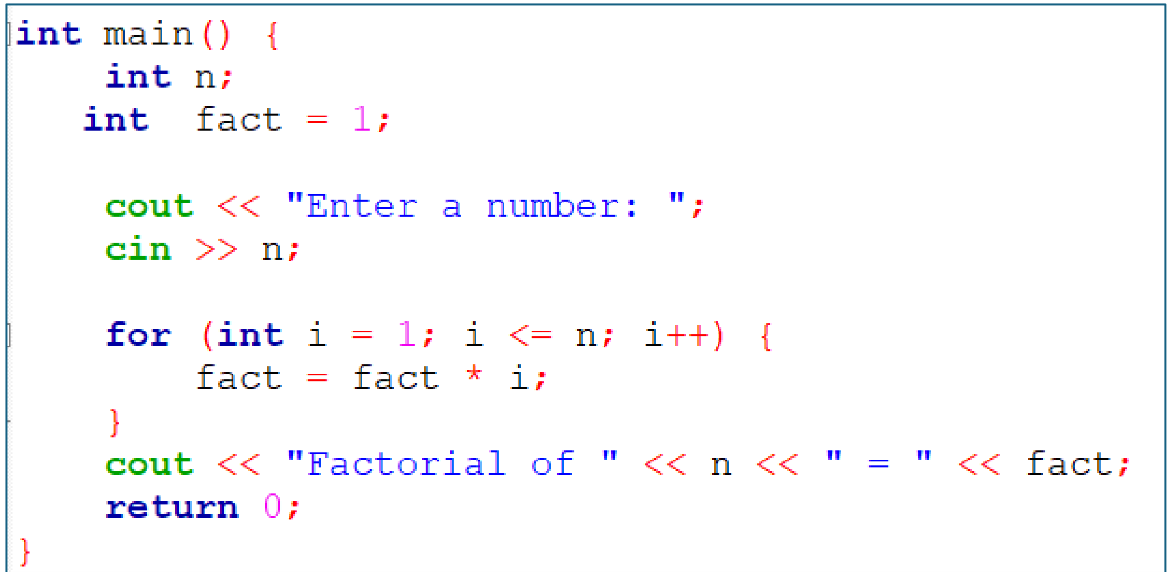
- **To Solve Problems That Are Naturally Recursive**

Many real-world problems can be expressed in terms of smaller versions of themselves.

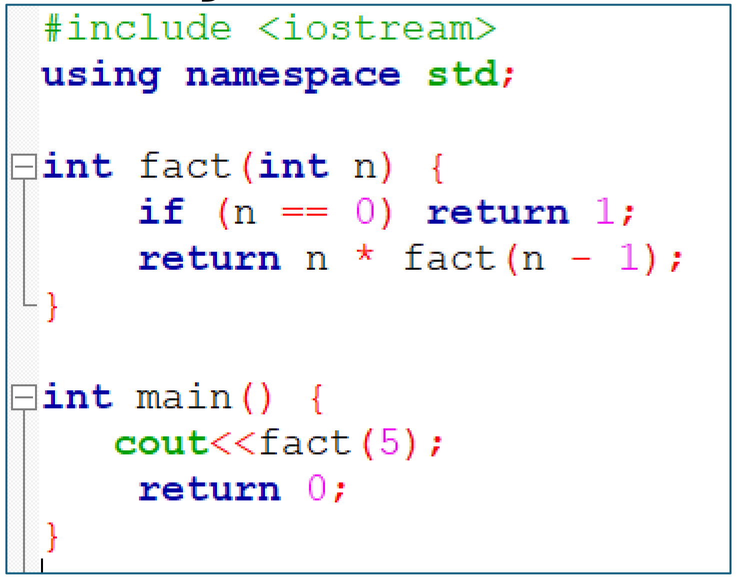
**Factorial:**

n! = n × (n-1)!

- **To Reduce Code Size and Improve Readability**



```
int main() {
     int n;
    int  fact = 1;

     cout << "Enter a number: ";
     cin >> n;

     for (int i = 1; i <= n; i++) {
         fact = fact * i;
     }
     cout << "Factorial of " << n << " = " << fact;
     return 0;
}
```



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

int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}

int main() {
   cout<<fact(5);
    return 0;
}
```

## Two parts of Recursion

- **Two Important Parts of Recursion**
- **Base Case:** Condition where recursion stops.
  Without it, recursion continues infinitely.
- **Recursive Case:** Function calls itself with modified arguments

## Write a program to find the **sum of first n natural numbers** using recursion.

Example: for n = 5, sum = 1 + 2 + 3 + 4 + 5 = 15

**Mathematically**

We know the mathematical relationship:

$$\text{sum}(n) = n + sum(n-1)$$

That means:

To find the sum of first 5 numbers → **5 + sum(4)**

To find sum(4) → **4 + sum(3)**

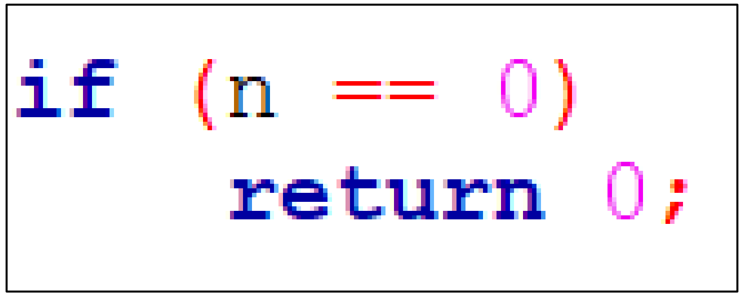
… and so on, until **sum(1) = 1**

# program to find the sum of first n natural numbers using recursion.

- Identify the Base Case – Every recursive function to stop at this case

Here, when n == 0, the sum is 0.

So base case:

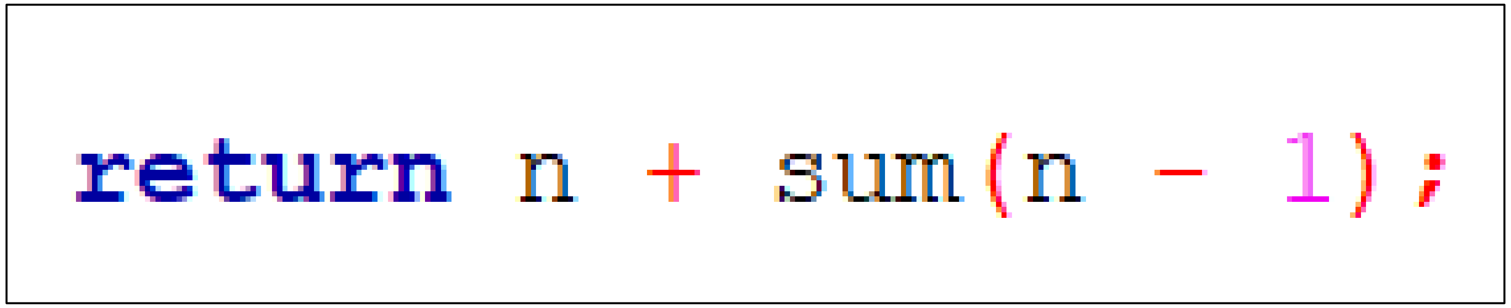


```
if (n == 0)
    return 0;
```

# program to find the sum of first n natural numbers using recursion

- Identify the recursive case:

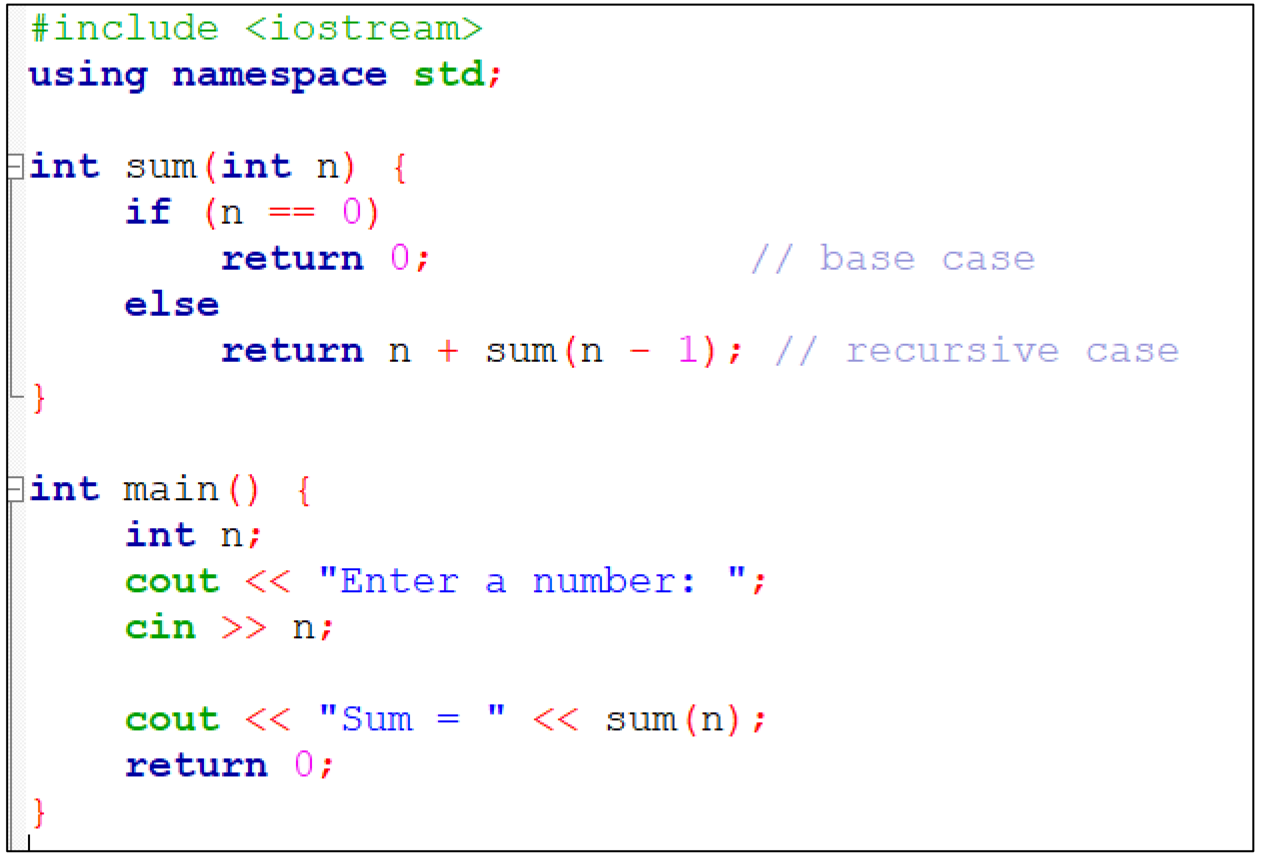
n>0



```
return n + sum(n - 1);
```

This means each call adds n to the sum of all numbers before it.

# program to find the sum of first n natural numbers using recursion



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

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

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;

    cout << "Sum = " << sum(n);
    return 0;
}
```

Example:
If n=3

sum(3) → 3 + sum(2)

sum(2) → 2 + sum(1)

sum(1) → 1 + sum(0)

sum(0) → 0 (base case)

Final output =6

# Program to find power of a number ($x^n$) using recursion

$$x^n = x * x^{(n-1)}$$

Base case - ?

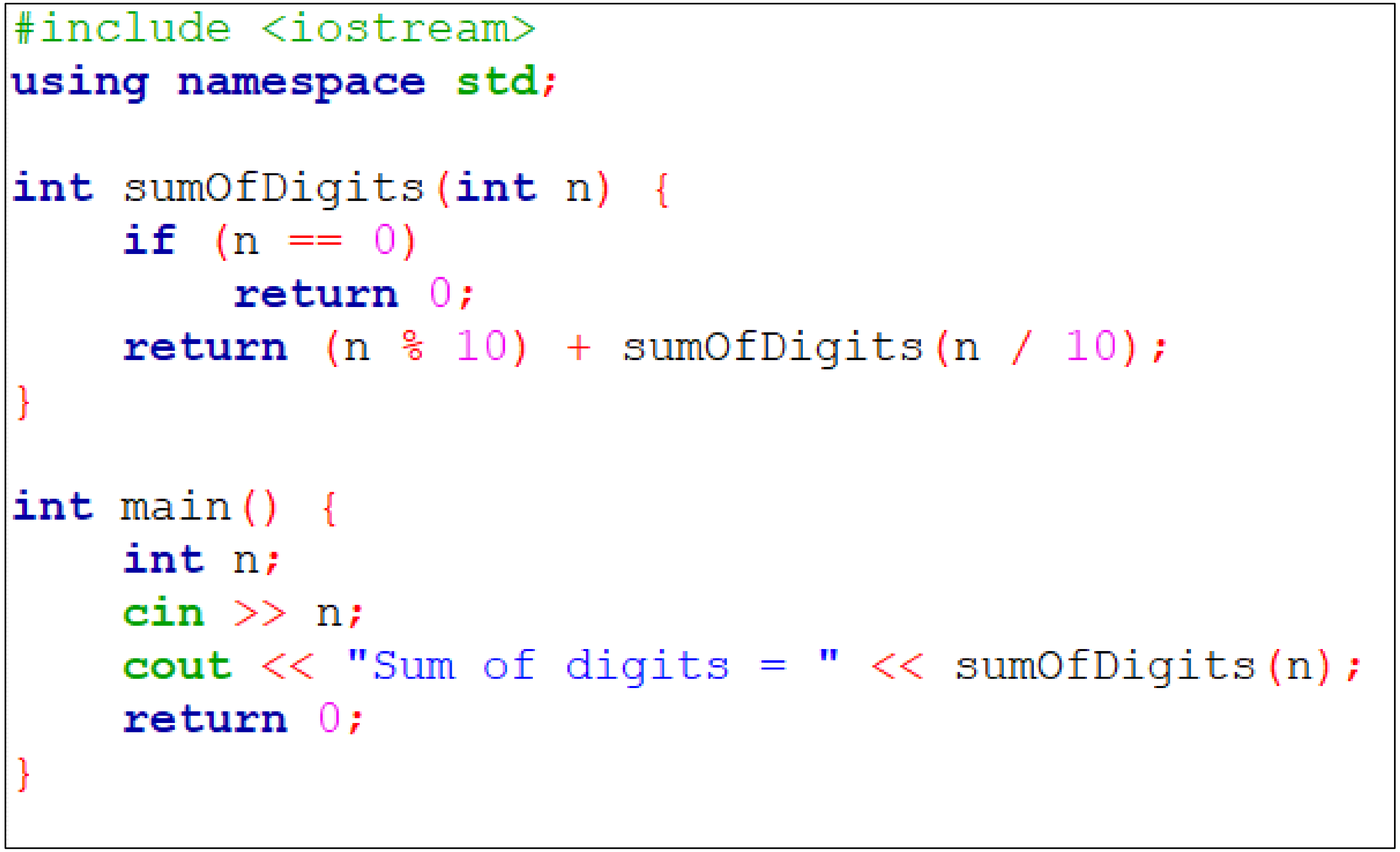
Recursive case -

```
Int powerx(int x, int n)
{
  if(n==0) return 1;
 return x*power(x,n-1);
}
X=2 n=4
Power(2,4) -> 2*8
```

## WAP to find the sum of digits of number using recursion

Sum(n)= n%10 + sum(n/10)

## Program sum of digits of number using recursion



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

int sumOfDigits(int n) {
    if (n == 0)
        return 0;
    return (n % 10) + sumOfDigits(n / 10);
}

int main() {
    int n;
    cin >> n;
    cout << "Sum of digits = " << sumOfDigits(n);
    return 0;
}
```

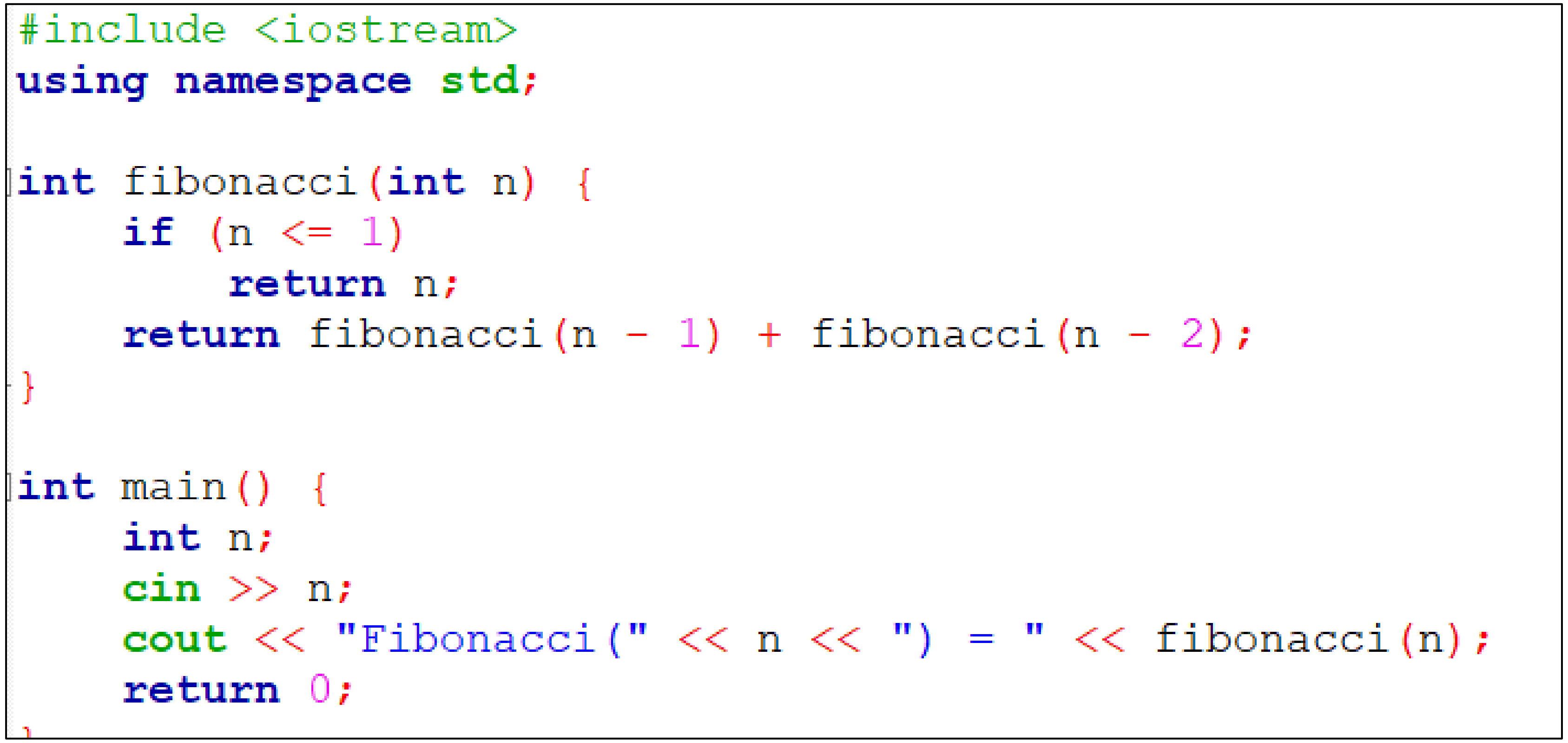
## Program to find nth Fibonacci Number

$$F(n) = F(n-1) + F(n-2)$$

Base case:

Recursive case:

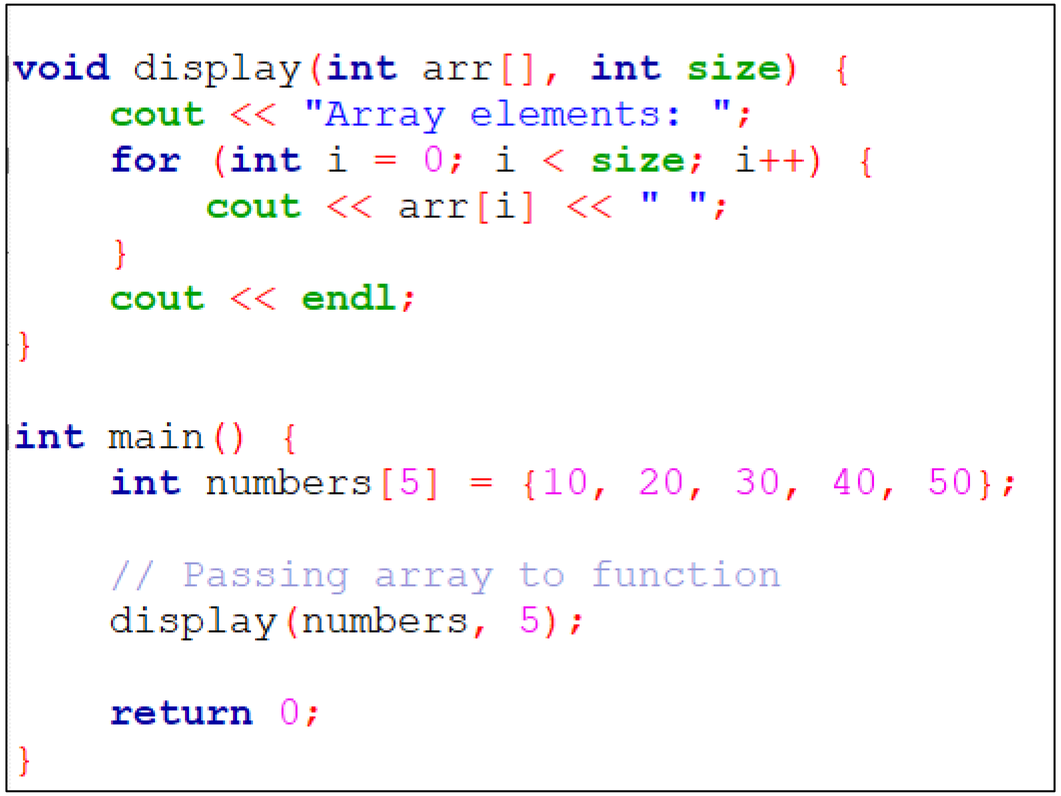
# Program to find nth Fibonacci number



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

int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

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

# Passing single dimension arrays to functions

- Arrays are always passed to a function by reference
- Base address of the array – address of first element in passed



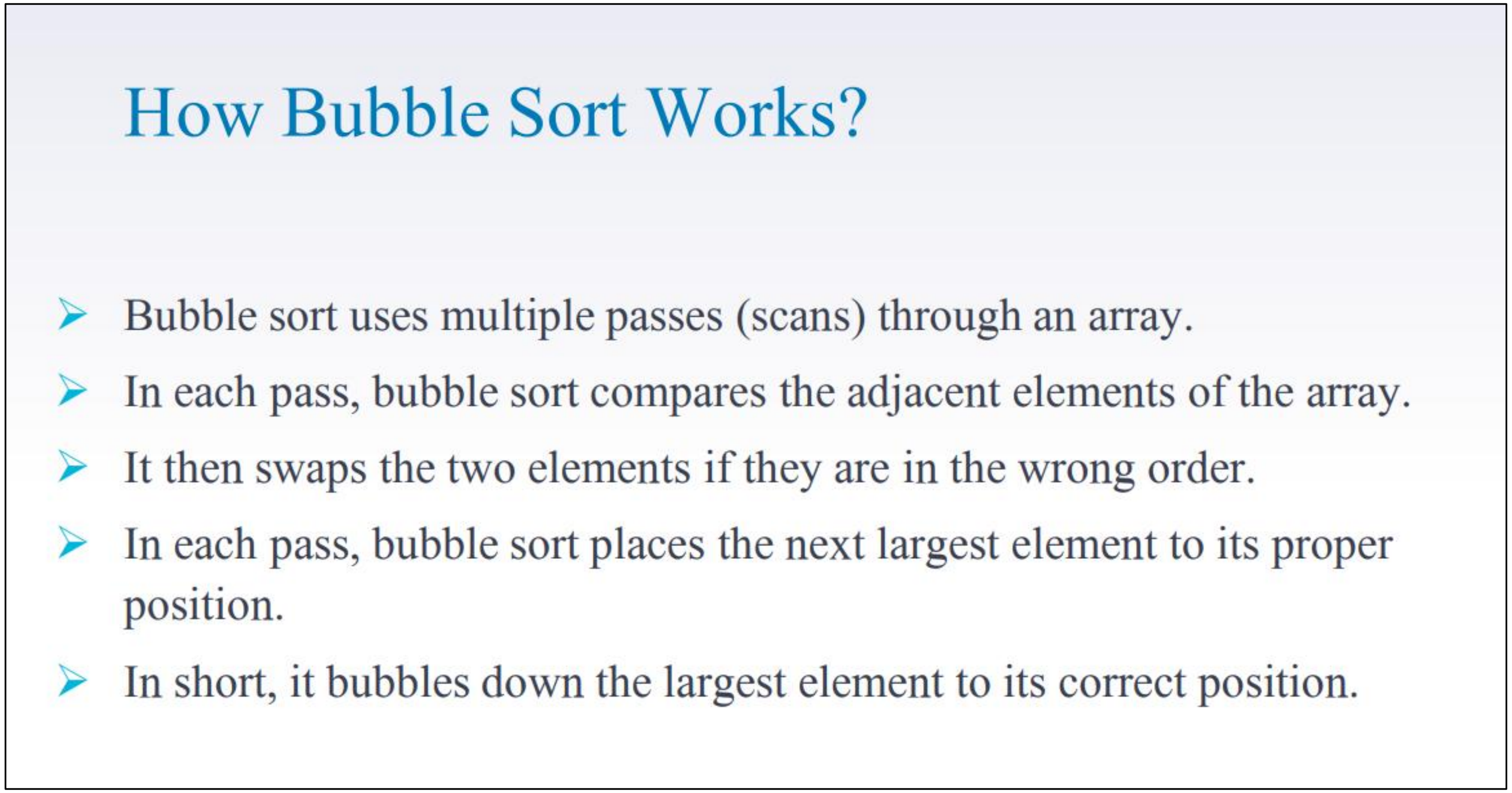
```
void display(int arr[], int size) {
    cout << "Array elements: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

    // Passing array to function
    display(numbers, 5);

    return 0;
}
```

**Explanation**
The array name numbers represents the **base address** (address of numbers[0]).

The function parameter int arr[] actually means int *arr.
So display(numbers, 5) is equivalent to display(&numbers[0], 5).

## Bubble sort

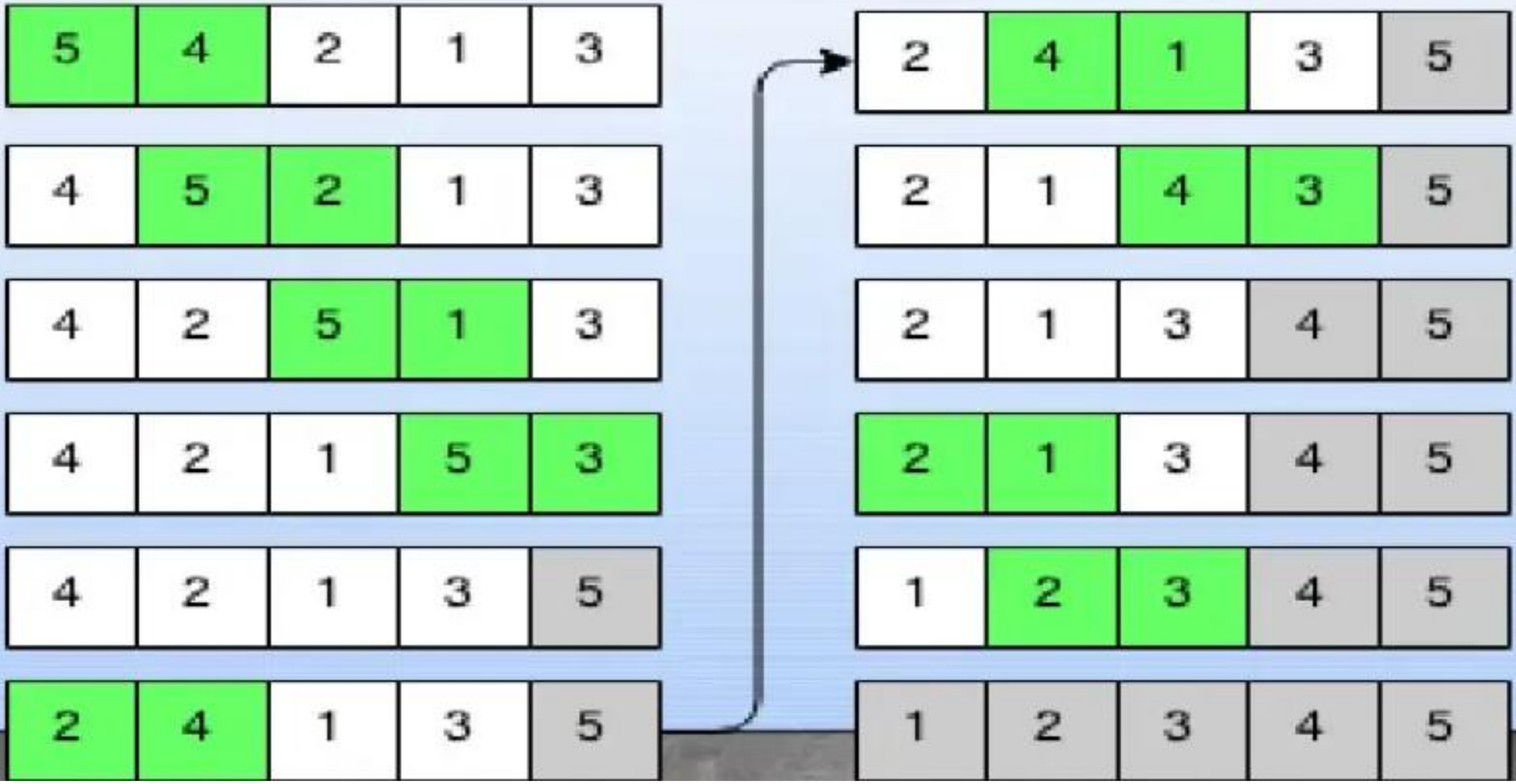
Bubble sort is simple sorting technique



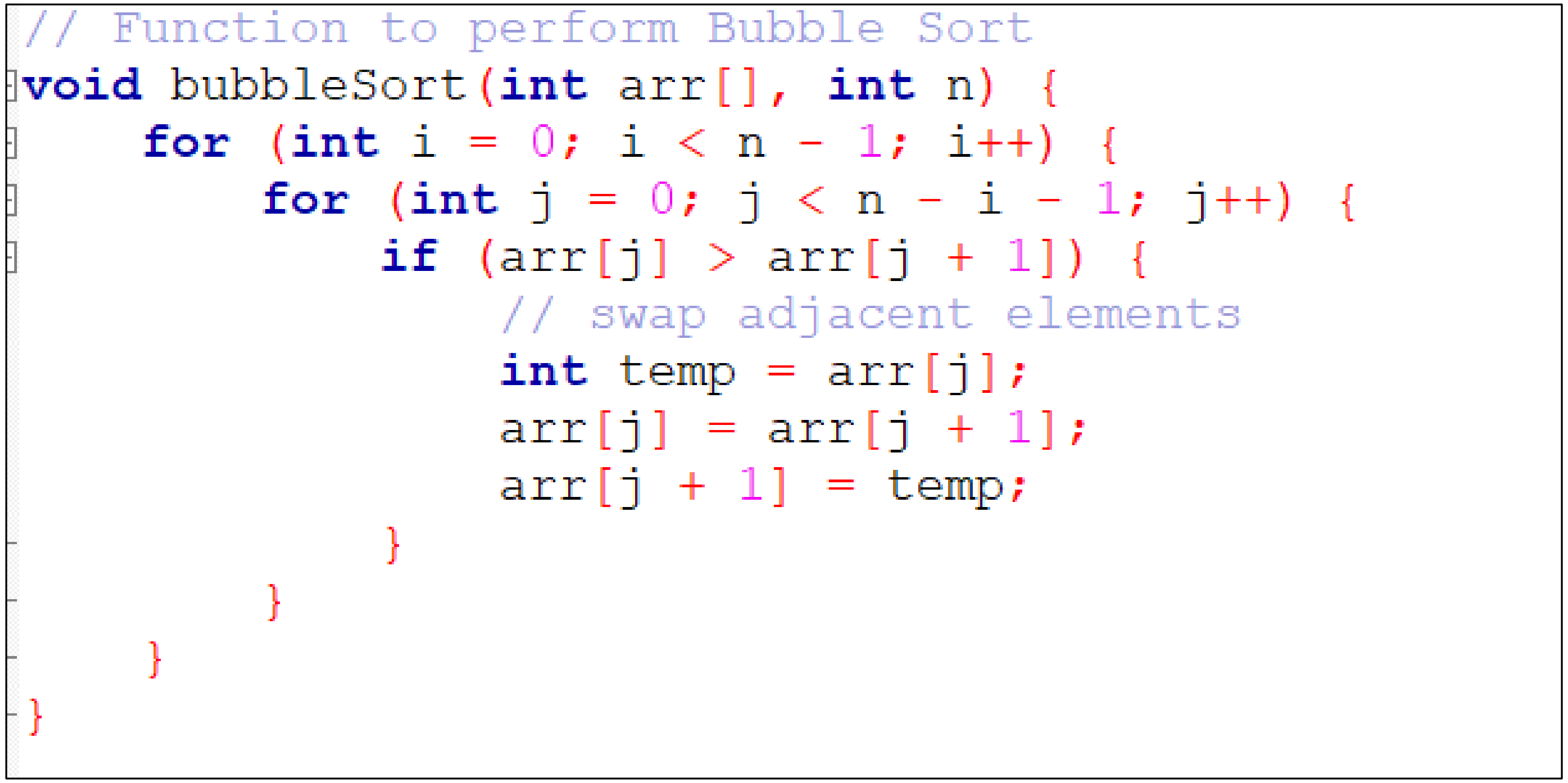
### How Bubble Sort Works?

- Bubble sort uses multiple passes (scans) through an array.
- In each pass, bubble sort compares the adjacent elements of the array.
- It then swaps the two elements if they are in the wrong order.
- In each pass, bubble sort places the next largest element to its proper position.
- In short, it bubbles down the largest element to its correct position.

An example for the bubble sort,



## Program to sort the array using bubble sort using functions



```
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap adjacent elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

N=4

12 , 45, 11, 29

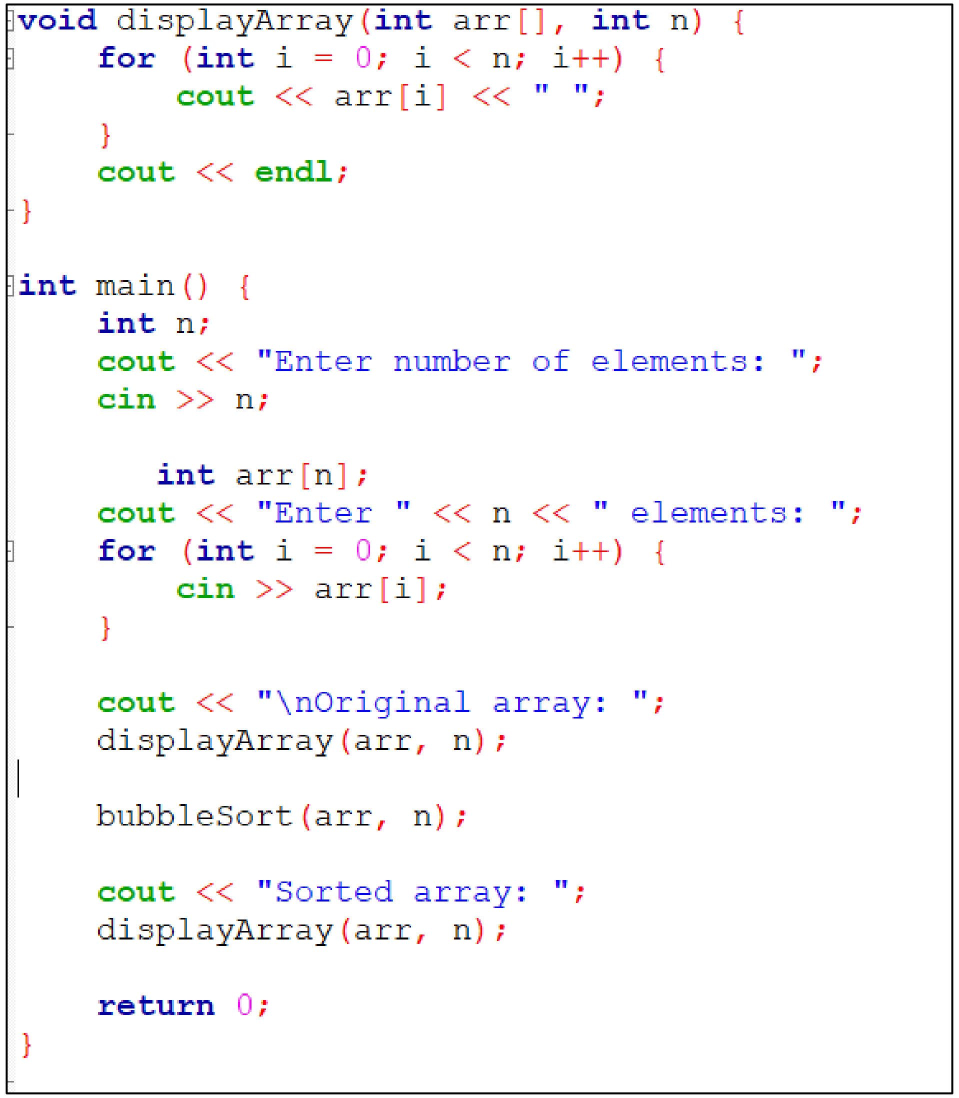
12,11,29, 45

11, 12, 29, 45

11, 12 , 29, 45

```
For(i=0;i<n;i++)
For(j=0;j<n-i-1;j++)
If(a[j]>a[j+1]) swap
```

## Main code

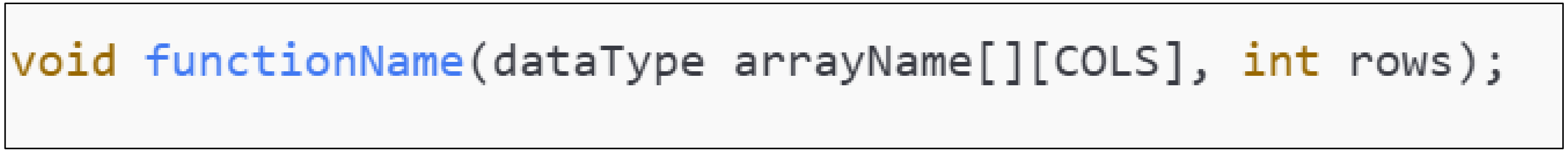


```
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

       int arr[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "\nOriginal array: ";
    displayArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    displayArray(arr, n);

    return 0;
}
```

## Passing 2D array to functions

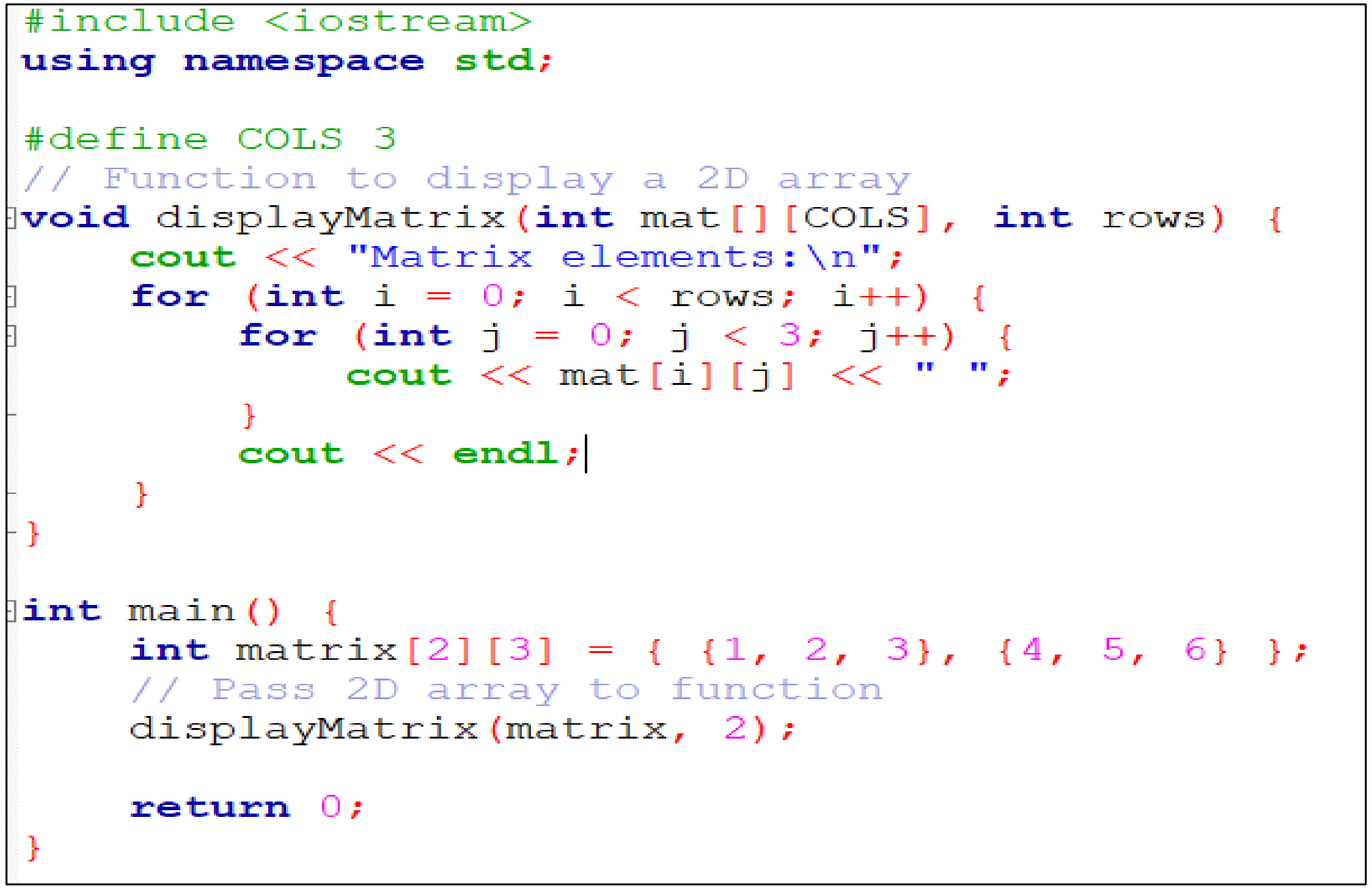
- When you pass a 2D array to a function, what's actually passed is the address of the first row — i.e., a pointer to an array. i.e arr[0][0]
- But  unlike 1D arrays — the compiler must know how many columns are in each row to correctly calculate element addresses.
- So, you must specify the number of columns in the function parameter (either as a constant,  or passed variable).

# define COLS 3



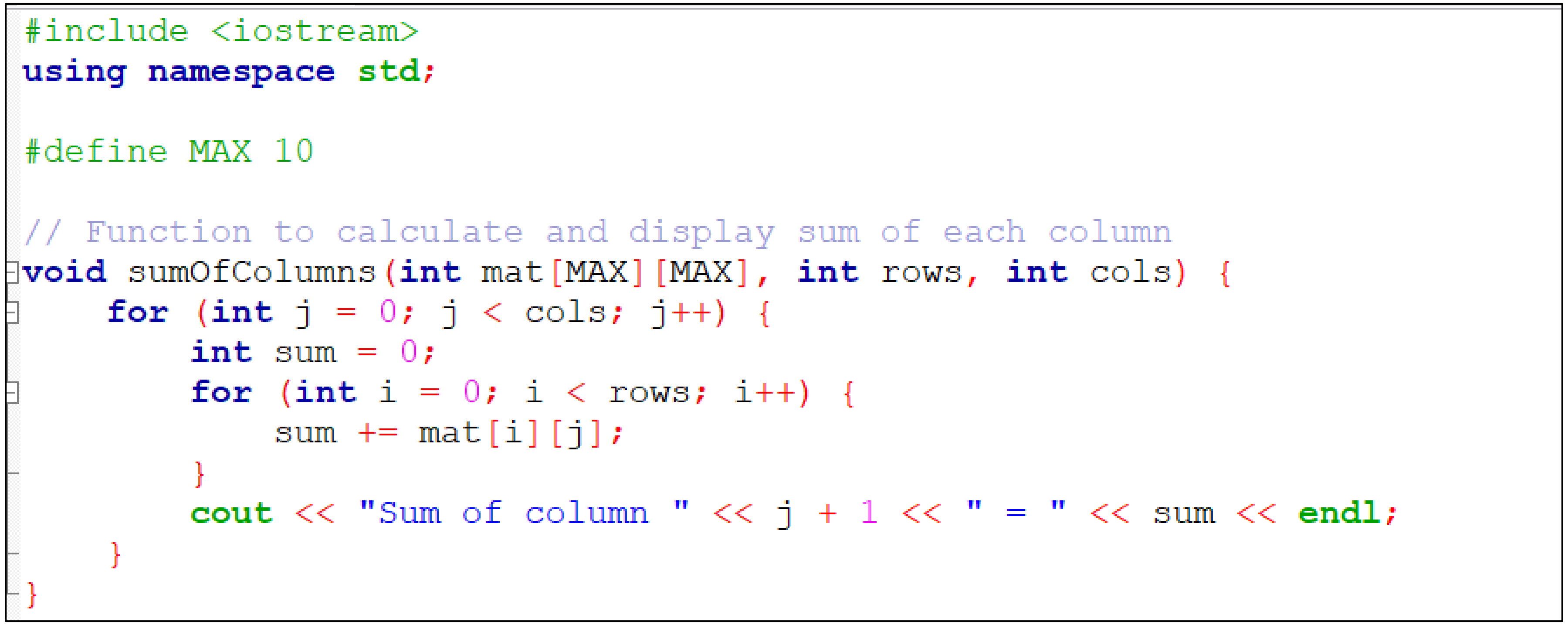
```
void functionName(dataType arrayName[][COLS], int rows);
```

## Example

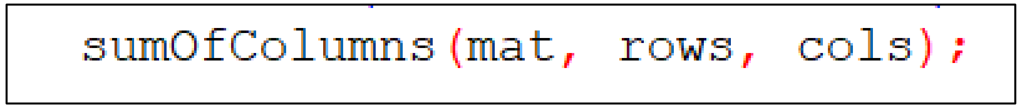


```
#include <iostream>
using namespace std;

#define COLS 3
// Function to display a 2D array
void displayMatrix(int mat[][COLS], int rows) {
    cout << "Matrix elements:\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            cout << mat[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
    // Pass 2D array to function
    displayMatrix(matrix, 2);

    return 0;
}
```

## WAP to display the sum of each column of matrix using recursion



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

#define MAX 10

// Function to calculate and display sum of each column
void sumOfColumns(int mat[MAX][MAX], int rows, int cols) {
    for (int j = 0; j < cols; j++) {
        int sum = 0;
        for (int i = 0; i < rows; i++) {
            sum += mat[i][j];
        }
        cout << "Sum of column " << j + 1 << " = " << sum << endl;
    }
}
```

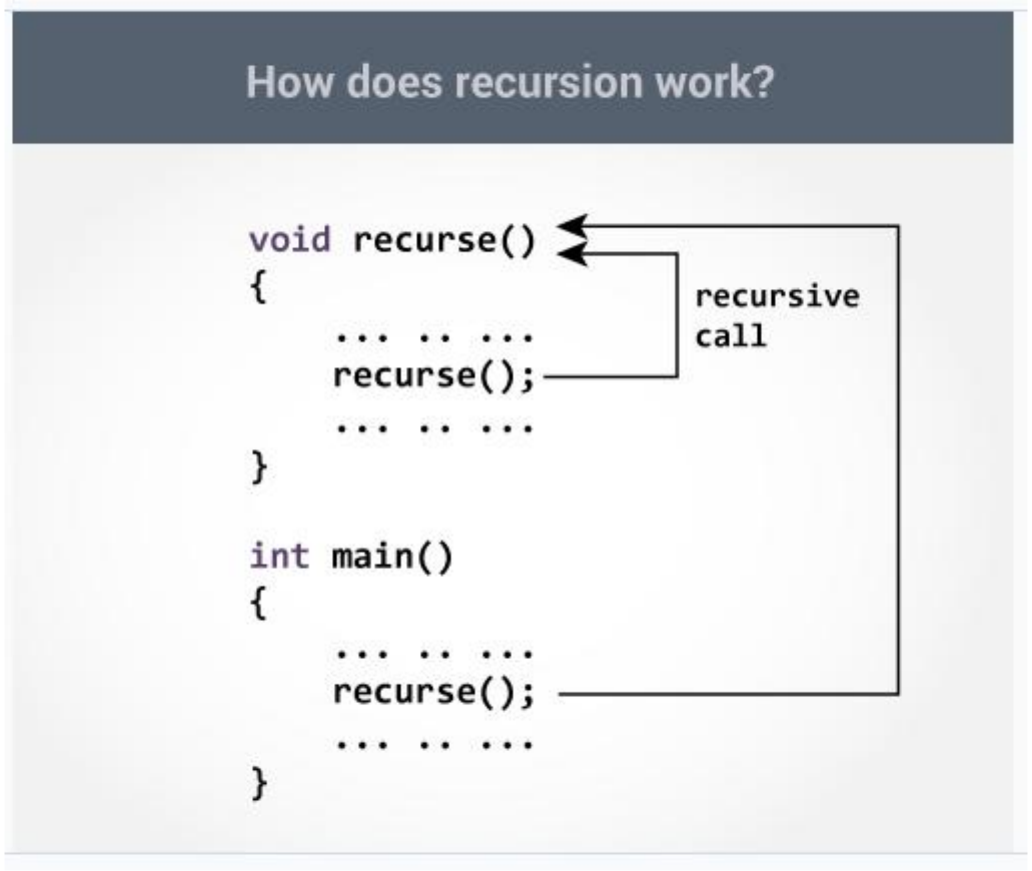


```cpp
sumOfColumns(mat, rows, cols);
```

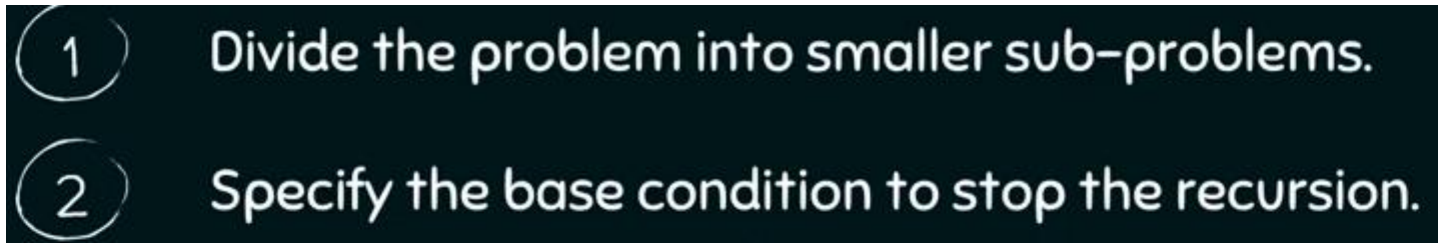
## Functions

## Function – Recursion.

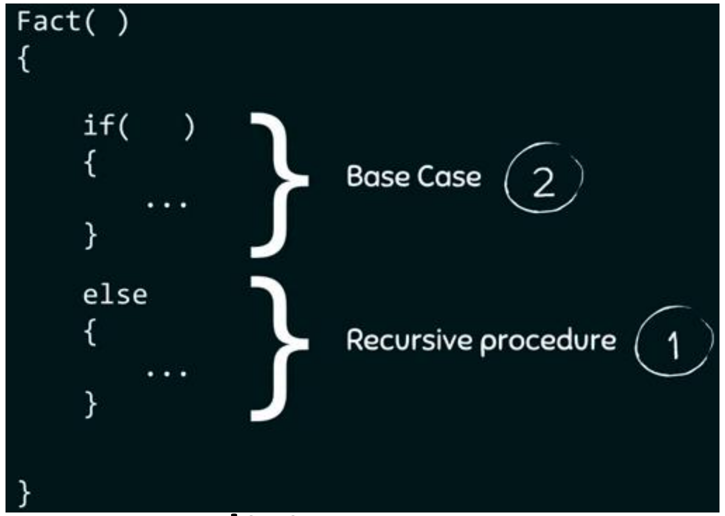
A function that calls itself is known as a recursive function. And, this technique is known as recursion.



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```



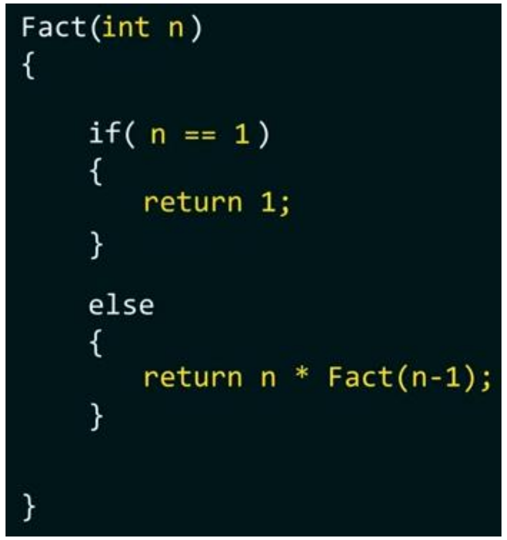
1. Divide the problem into smaller sub-problems.
2. Specify the base condition to stop the recursion.



```
Fact( )
{
    if(    )
    {
        ...
    }
    else
    {
        ...
    }
}
```

- Step 1: divide the problem
    - Fact(1)=1
    - Fact(2)=2*1
    - Fact(2)= 2* fact(1)
    - Fact(3)=3*2*1
    - Fact(3)= 3* fact(2)
    - Fact(2)=4*3*2*1
    - Fact(4)= 4* fact(3)
    - ...
    - Fact(n)= n* fact(n-1)

- Step2: base condition
- Fact(1)=1



```
Fact(int n)
{
    if( n == 1)
    {
        return 1;
    }
    else
    {
        return n * Fact(n-1);
    }
}
```

# Functions

## Function – Recursion---Example: Sum of Natural Numbers Using Recursion

```
#include <iostream>
using namespace std;
// Function Declaration
int sum(int n) {
    if (n != 0)
        return n + sum(n - 1);  // Recursive call
    else
        return 0;
}
int main() {
    int number, result;
    cout << "Enter a positive integer: ";
    cin >> number;
    result = sum(number);
    cout << "Sum = " << result << endl;
    return 0;
}
```

```
int sum(int n) {
    if (n != 0)
        return n + sum(n - 1);  //
Recursive call
    else
        return 0;
}
```

## Functions

## Task

**Find factorial of a number using recursion**

## Calclulate GCD of two numbers

**GCD of two numbers**

```
#include <iostream>
using namespace std;
int main() {
    int a, b, r = 0;
    cout << "Enter two numbers: ";
    cin >> a >> b;
    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }
    cout << "GCD is: " << a << endl;
    return 0;
```

**GCD of two numbers using function**

```
int gcd_num(int x, int y) {            // Function to calculate GCD
    int rem;
    while (y != 0) {
        rem = x % y;
        x = y;
        y = rem;
    }   return x;
}
int main() {
    int a, b, res;
    cout << "Enter two numbers: ";
    cin >> a >> b;
    res = gcd_num(a, b);
    cout << "GCD of " << a << " and " << b << " is " << res << endl;
    return 0;   }
```

## LCD of two numbers

- The formula to find the LCM of two numbers:

$$\mathrm{LCM}(a,b) = \frac{|a \times b|}{\mathrm{GCD}(a,b)}$$