MTE -2

CP

PADHLE BHAI.IN

Array Definition:

An array is a data structure that stores a collection of elements, each identified by an index or a key. These elements are typically of the same data type, and they are stored in contiguous memory locations. The primary purpose of an array is to organise and efficiently manage a set of related data.

```
#include <stdio.h>

int main() {
    // Declaration and initialization of an integer array
    int numbers[5] = {1, 2, 3, 4, 5};

    // Accessing and printing elements of the array
    for (int i = 0; i < 5; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }

    return 0;
}
```

Array declaration

Array declaration in text involves specifying the data type of the elements in the array and the array's name. If the size of the array is known at the time of declaration, it's also mentioned. The syntax for array declaration in text is generally as follows:

```
datatype arrayName[arraySize];
```

These declarations specify the type of data the array will hold, give the array a name, and, if applicable, indicate the size or dimensions of the array.

Array initialization

Array initialization in text involves providing initial values for the elements of the array at the time of declaration. The syntax for array initialization in text varies depending on the programming language. In C, array initialization is typically done using **curly braces {}.**

```
datatype arrayName[arraySize] = {value1, value2, ..., valueN};
```

Keep in mind that the array size (**arraySize**) in the declaration should match the number of elements provided in the initialization list. If the size is omitted, the compiler will infer it from the number of elements in the list.

Accessing Arrays

Accessing elements in an array involves using the array name followed by an index enclosed in square brackets. The index specifies the position of the element in the array. In many programming languages, including C, array indexing starts at 0.

```
arrayName[index]
```

Remember that array indices start at 0, so the first element is accessed with index 0, the second with index 1, and so on. Be cautious not to access elements beyond the bounds of the array to prevent undefined behaviour and potential errors in your program.

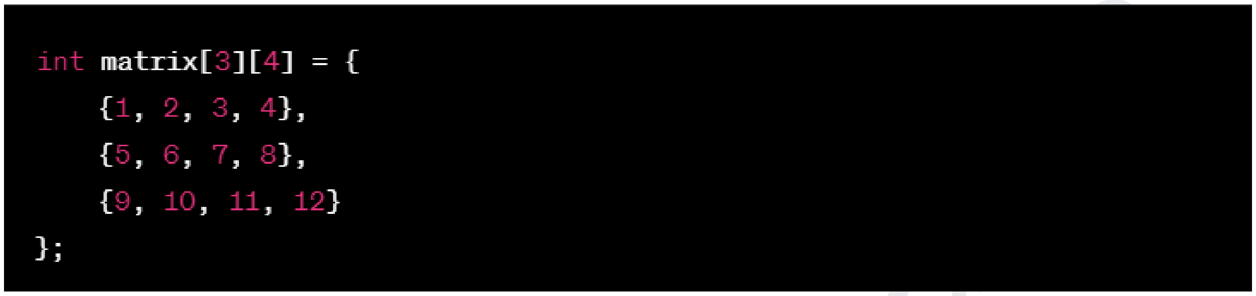
One-Dimensional Array (1D Array):

- A one-dimensional array is a linear collection of elements, and each element is identified by a single index.
- **Declaration in C:** datatype arrayName[size];

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

Two-Dimensional Array (2D Array):

A two-dimensional array is a table of elements with rows and columns, and each element is identified by two indices (row index and column index).
Declaration in C: datatype arrayName[rows][columns];



```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Pointer:

Pointer is a variable that stores the memory address of another variable. Pointers are a fundamental concept in languages like C and C++, and they allow for more direct and efficient manipulation of memory. Here are some key points about pointers:

Memory Address:

- A pointer is essentially a variable that contains the memory address of another variable.
- The value of a pointer is the memory address it points to.

Declaration:

- In C, a pointer is declared using the * (asterisk) symbol.
- Example: int *ptr; declares a pointer variable named ptr that can point to an integer.

Initialization:

- A pointer can be initialised with the address of a variable.
- Example: int x = 10; int *ptr = &x; initialises ptr with the address of the integer variable x.

Dereferencing:

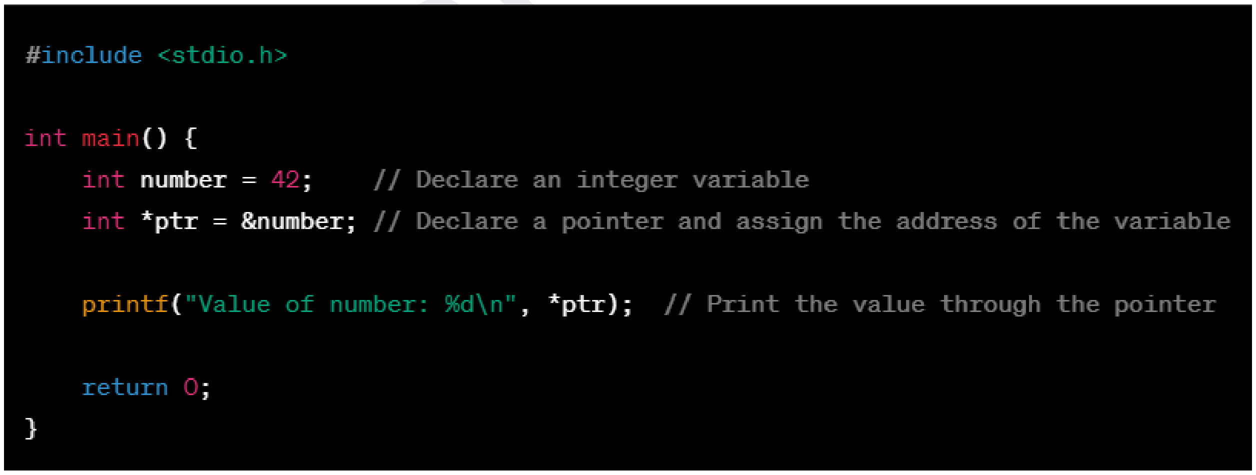
- Dereferencing a pointer means accessing the value stored at the memory address it points to.
- Example: int value = *ptr; retrieves the value at the memory address stored in ptr.

Pointer Arithmetic:

- Pointers can be incremented or decremented to point to the next or previous memory location.
- Example: ptr++; increments the pointer to point to the next memory location.

NULL Pointer:

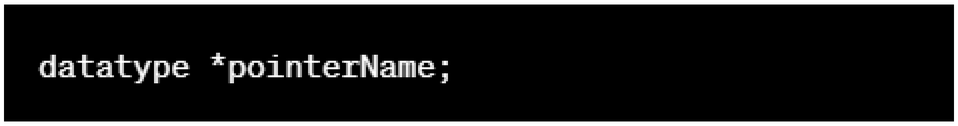
- A pointer that does not point to any valid memory address is called a NULL pointer. It is typically assigned the value NULL (or 0 in C).
- Example: int *ptr = NULL;



```c
#include <stdio.h>

int main() {
    int number = 42;    // Declare an integer variable
    int *ptr = &number; // Declare a pointer and assign the address of the variable

    printf("Value of number: %d\n", *ptr);  // Print the value through the pointer

    return 0;
}
```

Declaration of Pointer:

In C, the declaration of a pointer involves specifying the data type of the variable it will point to, followed by an asterisk **(*)**, and then the name of the pointer. Here's the basic syntax:



```
datatype *pointerName;
```

Here, **datatype** is the type of data that the pointer will point to, and **pointerName** is the chosen name for the pointer.

Applications of Pointers:

Dynamic Memory Allocation:

- Pointers are commonly used to allocate memory dynamically during program execution, enabling flexibility in memory management.

Function Pointers:

- Pointers can be used to store the address of functions, allowing for dynamic invocation of different functions during runtime.

Passing Parameters by Reference:

- Pointers enable passing parameters by reference to functions, allowing functions to modify the original values of variables.

Arrays and Strings Manipulation:

- Pointers facilitate efficient traversal and manipulation of arrays and strings, providing a way to access and modify elements directly in memory.

Function:

A function is a self-contained block of code that performs a specific task or set of tasks. Functions are designed to be reusable, modular components that enhance code organisation and maintainability. The fundamental purpose of functions is to break down a program into smaller, manageable pieces, each responsible for a specific functionality.

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

Declaration of a Function :

The declaration of a function in text involves specifying the function's name, return type, and the types of its parameters (if any). Here's the basic syntax for a function declaration:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

In this syntax:

- **return_type** is the data type of the value that the function returns.
- **function_name** is the chosen name for the function.
- **parameter_type1**, **parameter_type2**, etc., are the data types of the parameters the function accepts (if any).

Types of Functions:

Built-in Functions:

- These functions are provided by the programming language or its standard library. Examples include printf and scanf in C, and print and input in Python.

User-Defined Functions:

- Functions created by the programmer to perform specific tasks. They help in modularizing code and improving code readability.

Recursive Functions:

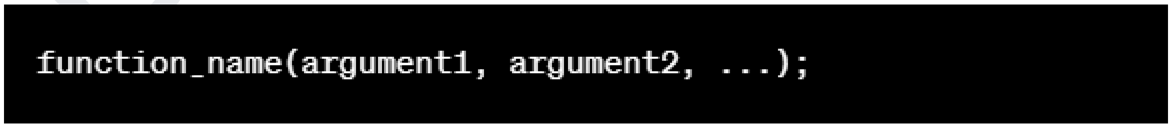
- Functions that call themselves, either directly or indirectly. Recursion is often used for solving problems that can be broken down into smaller subproblems.

Library Functions:

- Functions that are not part of the standard library but are available in external libraries. These can include functions for mathematical operations, file handling, and more.

Calling a Function:

Calling a function in a programming language involves using the function's name followed by parentheses, and providing any required arguments within the parentheses. Here is a general syntax for calling a function:



```
function_name(argument1, argument2, ...);
```

```c
#include <stdio.h>

// Function declaration
void sayHello();

int main() {
    // Function call
    sayHello();

    return 0;
}

// Function definition
void sayHello() {
    printf("Hello, World!\n");
}
```

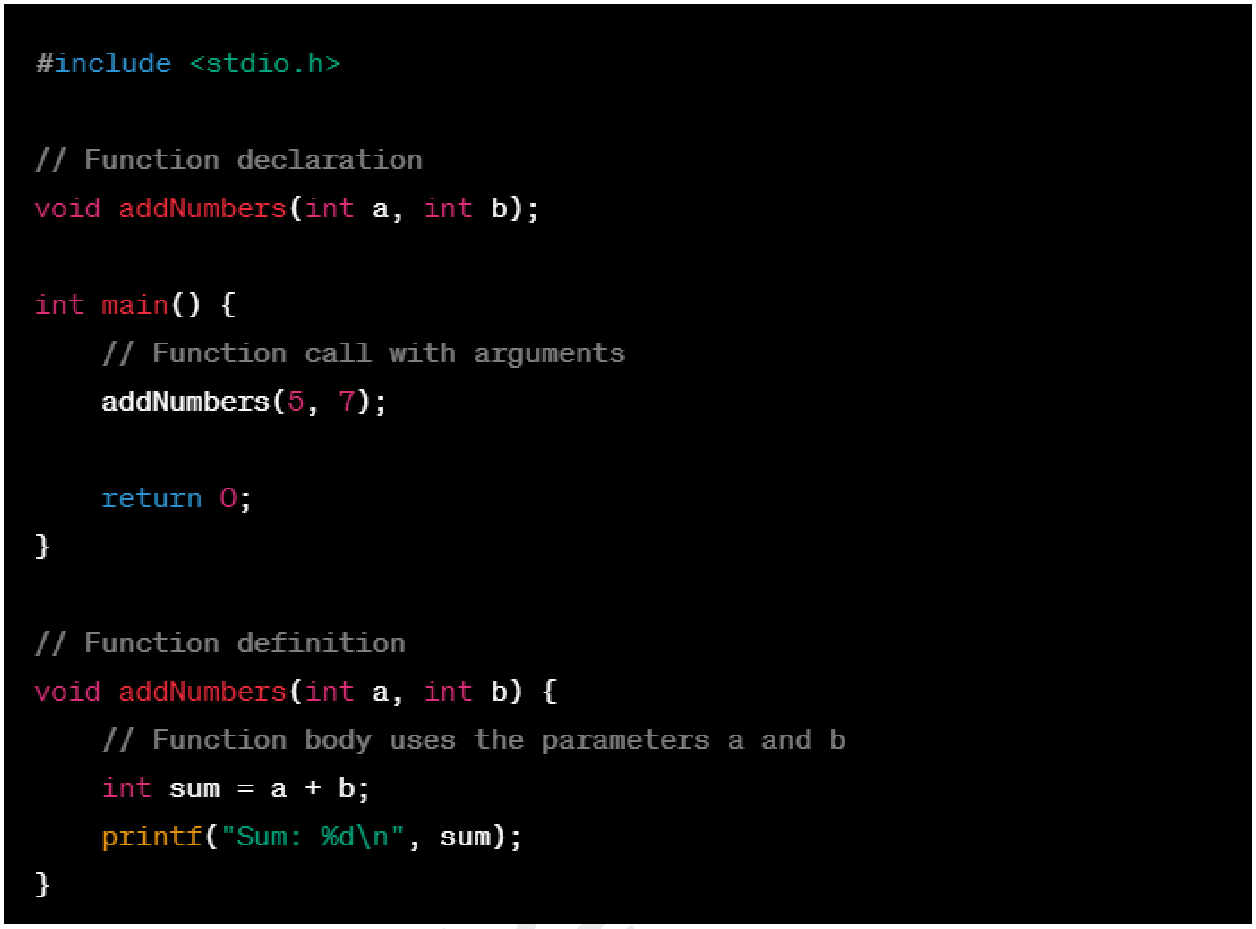
In this example:

The function **sayHello** is declared and defined to print **"Hello, World!"** to the console.
In the main function, the sayHello function is called.
The program prints **"Hello, World!"** when executed.

Function arguments:

Function arguments, also known as parameters, are values that are passed to a function when it is called. Functions use these parameters to perform specific tasks or calculations. Function arguments allow you to provide input to a function, enabling it to work with different data each time it is called. Here's a basic overview:

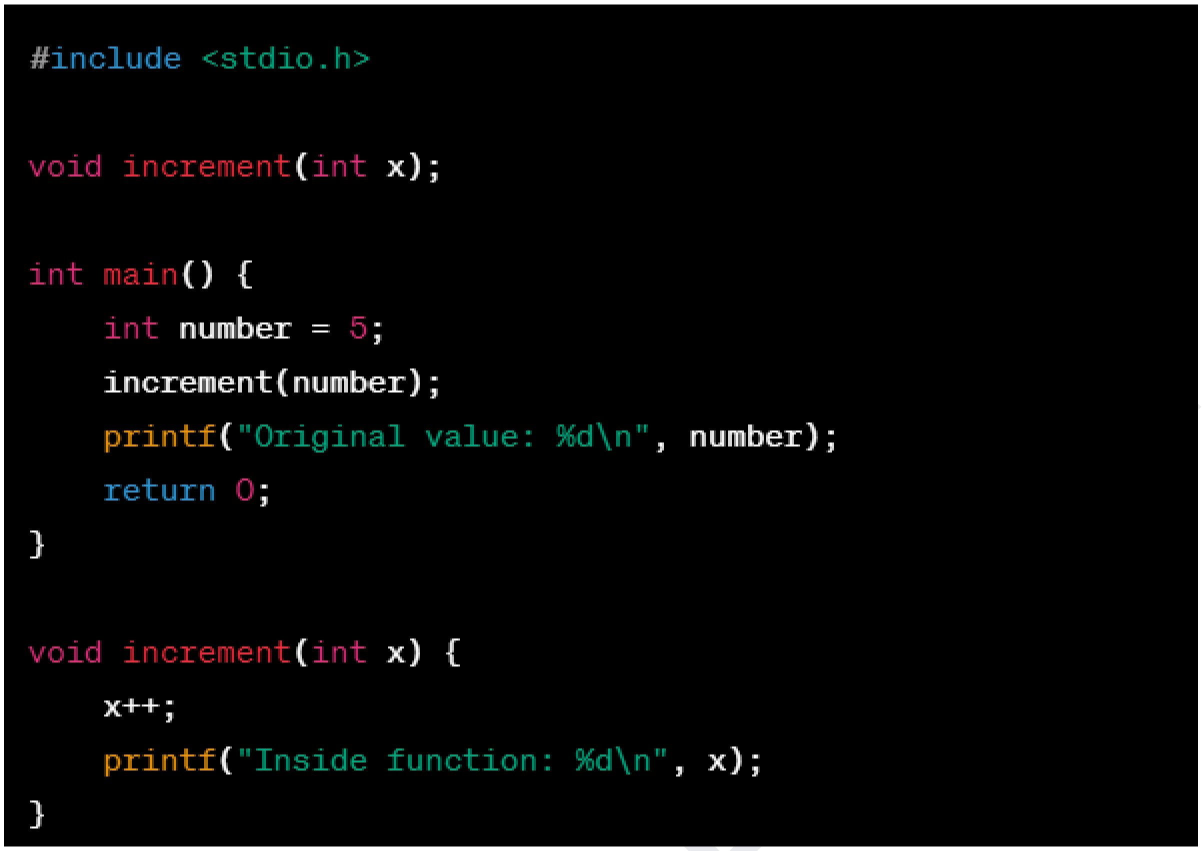
```c
return_type function_name(data_type parameter1, data_type parameter2, ...) {
    // Function body
}
```



```c
#include <stdio.h>

// Function declaration
void addNumbers(int a, int b);

int main() {
    // Function call with arguments
    addNumbers(5, 7);

    return 0;
}

// Function definition
void addNumbers(int a, int b) {
    // Function body uses the parameters a and b
    int sum = a + b;
    printf("Sum: %d\n", sum);
}
```

Call by value:

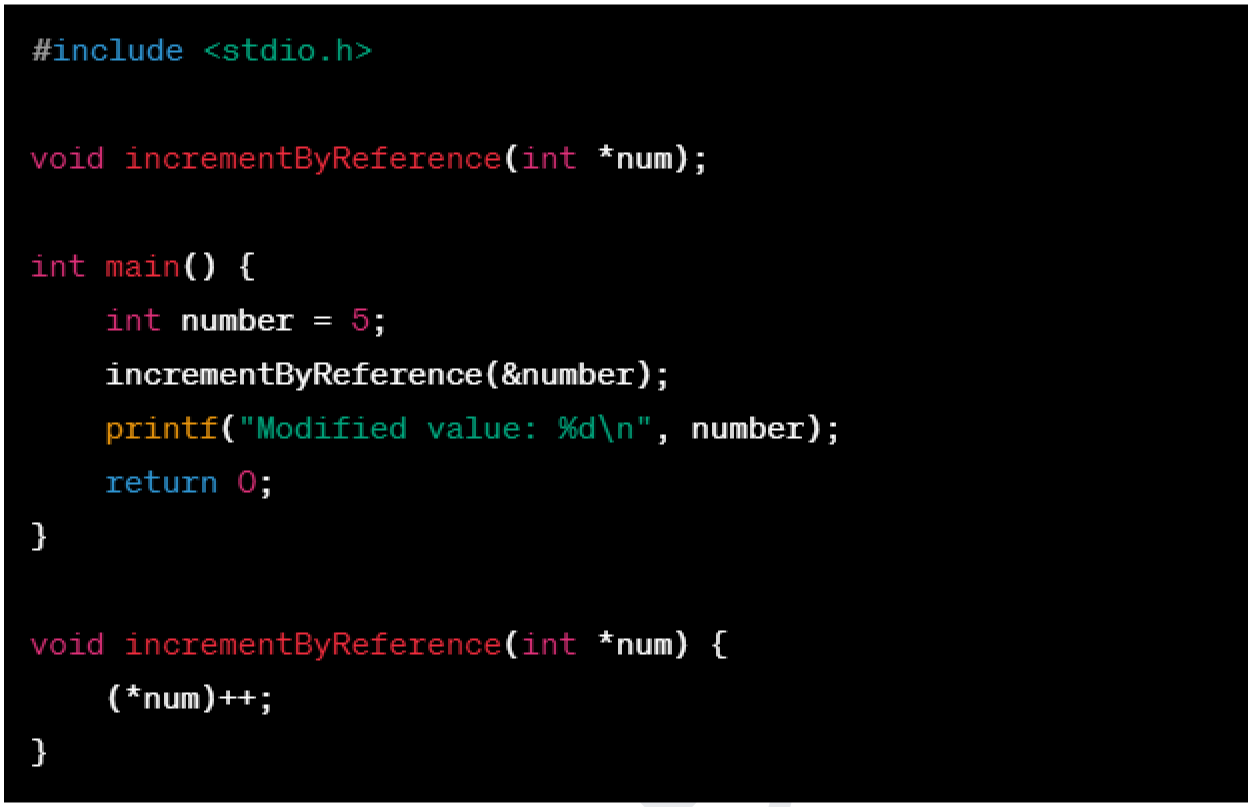
In call by value, when you pass arguments to a function, the values of the arguments are copied into the function parameters. Any modifications made to the parameters inside the function do not affect the original values in the calling code. This is the default method of parameter passing in most programming languages.



```c
#include <stdio.h>

void increment(int x);

int main() {
    int number = 5;
    increment(number);
    printf("Original value: %d\n", number);
    return 0;
}

void increment(int x) {
    x++;
    printf("Inside function: %d\n", x);
}
```

Call by reference:

In call by reference, the memory address of the actual arguments (variables) is passed to the function parameters. This means that any changes made to the parameters inside the function directly affect the original variables in the calling code. In languages like C, this is typically achieved using pointers.



```c
#include <stdio.h>

void incrementByReference(int *num);

int main() {
    int number = 5;
    incrementByReference(&number);
    printf("Modified value: %d\n", number);
    return 0;
}

void incrementByReference(int *num) {
    (*num)++;
}
```

MTE -2

CP

PADHLE BHAI.IN