Local Variable vs Global Variable in C

1. Local Variable

- Declared **inside a function or a block { }**.
- Scope is **limited to that function/block only**.
- Created when the function is called and destroyed when the function ends.
- **Default value is garbage** (if not initialized).
- Stored in the **stack memory**.

Example:

```
#include <stdio.h>

void myFunction() {
    int x = 10;  // Local variable
    printf("x = %d\n", x);
}

int main() {
    myFunction();
    // printf("%d", x);  // Error: x not accessible here
    return 0;
}
```

2. Global Variable

- Declared **outside all functions** (at the top level).
- Scope is **throughout the program** (accessible by all functions).
- Exists for the **entire lifetime of the program**.
- **Default value is 0** (if not initialized).
- Stored in the **data segment memory**.

Example:

```
#include <stdio.h>

int g = 100;  // Global variable

void myFunction() {
    printf("g in myFunction = %d\n", g);
}

int main() {
    printf("g in main = %d\n", g);
    myFunction(); // Accessible inside another function too
    return 0;
}
```

3. Key Differences

Feature	Local Variable	Global Variable
Declaration	Inside function/block	Outside all functions (top-level)
Scope	Only within that function/block	Entire program
Lifetime	Created on function call, destroyed on function exit	Exists till program ends
Default Value	Garbage (undefined)	0
Memory Allocation	Stack	Data Segment

Parameters and Arguments in C

When we use functions, we pass **data** into them. This involves **parameters** and **arguments**.

1. Actual Parameter (Arguments)

- The values or variables that are **passed to a function** when it is called.
- These exist in the **calling function**.
- They are also called **arguments**.

Example:

```
#include <stdio.h>

void display(int x) {   // x = Formal parameter
    printf("Value: %d\n", x);
}

int main() {
    int num = 10;
    display(num);  // num is Actual parameter (Argument)
    return 0;
}
```

2. Formal Parameter (Local Parameter)

- The variables that are **declared in the function definition**.
- They receive the values from actual parameters.
- They behave like **local variables inside the function**.

In the example above:

- `int x` inside `display(int x)` is the **Formal Parameter**.

3. Difference Between Actual and Formal Parameters

Feature	Actual Parameter (Argument)	Formal Parameter (Local Parameter)
Where defined	In the **function call**	In the **function definition**
Value	Provides the value	Receives the value

Feature	Actual Parameter (Argument)	Formal Parameter (Local Parameter)
Memory location	Exists in **calling function**	Exists in **called function**
Type	Must match data type of formal	Must match data type of actual

4. Example with Multiple Parameters

```
#include <stdio.h>

// Formal parameters: a, b
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 7;
    // Actual parameters: x, y
    int sum = add(x, y);
    printf("Sum = %d\n", sum);
    return 0;
}
```

- **Actual Parameters** → `x, y` (values passed when calling `add(x, y)`)
- **Formal Parameters** → `a, b` (local variables inside `add()` that receive the values).

Arrays in C Programming

1. Definition

- An **array** is a collection of **elements of the same data type** stored at **contiguous memory locations**.
- Each element can be accessed using an **index**.
- Indexing in C starts from **0**.

2. Syntax

```
data_type array_name[size];
```

Example:

```
int marks[5];   // An array of 5 integers
```

3. Initialization

Arrays can be initialized in different ways:

```
int a[5] = {10, 20, 30, 40, 50};   // Fully initialized
int b[5] = {1, 2};                 // Partially initialized, rest will be 0
int c[] = {5, 10, 15};             // Size auto-calculated (size = 3)
```

4. Accessing Elements

- Elements are accessed using **index number**.
- Index starts from `0` to `size-1`.

```
#include <stdio.h>
int main() {
    int arr[3] = {10, 20, 30};
    printf("%d", arr[0]); // Output: 10
    printf("%d", arr[2]); // Output: 30
    return 0;
}
```

5. Types of Arrays

1. One-Dimensional Array

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

2. Two-Dimensional Array (Matrix)

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

3. Multi-Dimensional Array

```
int arr[2][2][3]; // 3D array
```

6. Array with Loops

```
#include <stdio.h>
int main() {
    int i, arr[5] = {10, 20, 30, 40, 50};

    for(i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output: `10 20 30 40 50`

7. Important Points

- Array size must be a **constant integer** in C.
- If you try to access an index **outside the range**, it may cause **undefined behavior**.
- Arrays can be passed to **functions**.

Pointers in C

- A **pointer** is a variable that stores the **memory address** of another variable.
- Declared using `*` (asterisk).

Syntax:

```
data_type *pointer_name;
```

Example:

```
int x = 10;
int *p = &x;   // p stores address of x
```

2. Pointer Operators

- `&` → Address-of operator (gives address).
- `*` → Value-at operator (dereference, gives value at that address).

Example:

```
#include <stdio.h>
int main() {
    int x = 5;
    int *ptr = &x;
    printf("Address of x = %p\n", ptr);
    printf("Value of x using pointer = %d\n", *ptr);
    return 0;
}
```

Pointer with Arrays

1. Relation Between Array and Pointer

- Array name itself acts as a **pointer to the first element**.
- `arr` = address of `arr[0]`.

Example:

```
#include <stdio.h>
int main() {
    int arr[3] = {10, 20, 30};
    printf("arr = %p\n", arr);      // Address of first element
    printf("&arr[0] = %p\n", &arr[0]); // Same as arr
    return 0;
}
```

2. Accessing Array Elements using Pointer

- `arr[i]` is same as `*(arr + i)`.

Example:

```
#include <stdio.h>
int main() {
    int arr[3] = {10, 20, 30};
    int *ptr = arr;   // pointer to first element

    printf("First element: %d\n", *ptr);       // 10
    printf("Second element: %d\n", *(ptr+1));  // 20
    printf("Third element: %d\n", *(ptr+2));   // 30

    return 0;
}
```

3. Traversing Array using Pointer

```
#include <stdio.h>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr;

    for(int i = 0; i < 5; i++) {
        printf("%d ", *(ptr + i));
    }
    return 0;
}
```

Output: `1 2 3 4 5`

4. Pointer Arithmetic

- If `p` is an integer pointer:
    - `p + 1` → moves to the **next integer** (4 bytes ahead in most systems).
- Works similarly for other data types (depends on size).

Example:

```
#include <stdio.h>
int main() {
    int arr[3] = {10, 20, 30};
    int *p = arr;
    printf("%d\n", *p);     // 10
    printf("%d\n", *(p+1)); // 20
    printf("%d\n", *(p+2)); // 30
    return 0;
}
```

Recursion in C

- **Recursion** is a process where a **function calls itself** directly or indirectly.
- Every recursive function must have a **base condition** to stop further calls.

(a) Factorial using Recursion

```
#include <stdio.h>
int factorial(int n) {
    if(n == 0)      // Base case
        return 1;
    else
        return n * factorial(n-1); // Recursive case
}
int main() {
    int num = 5;
    printf("Factorial = %d", factorial(num));
    return 0;
}
```

Output: `Factorial = 120`

(b) Fibonacci using Recursion

```
#include <stdio.h>
int fibonacci(int n) {
    if(n == 0) return 0;    // Base case
    if(n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2); // Recursive case
}
int main() {
    for(int i = 0; i < 6; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```

Output: `0 1 1 2 3 5`

5. How Recursion Works (Execution Flow)

- Recursion uses **stack memory**.
- Each function call is stored on the call stack until the base case is reached.
- After the base case, functions return in **reverse order** (stack unwinding).

6. Advantages

- Reduces code size (short and clean programs).
- Useful for problems like factorial, Fibonacci, Tower of Hanoi, tree traversal, etc.

7. Disadvantages

- Uses more memory (stack calls).
- Slower execution than loops for large inputs.
- Risk of **stack overflow** if base condition is missing.

Call by Value vs Call by Reference in C

1. Call by Value

- **Definition**: A copy of the actual parameter (argument) is passed to the function.
- Changes made inside the function **do not affect the original variable**.
- Default method in C.

Example:

```
#include <stdio.h>
void change(int x) {
    x = 20;   // Only local copy changes
}
int main() {
    int a = 10;
    change(a);
    printf("a = %d", a);  // Output: 10 (unchanged)
    return 0;
}
```

2. Call by Reference

- **Definition**: Instead of value, the **address (reference)** of the variable is passed.
- Changes made inside the function **affect the original variable**.
- In C, this is done using **pointers**.

Example:

```
#include <stdio.h>
void change(int *x) {
    *x = 20;   // Changes the actual variable
}
int main() {
    int a = 10;
    change(&a);   // Passing address
    printf("a = %d", a);  // Output: 20 (changed)
    return 0;
}
```

3. Key Differences

Feature	Call by Value	Call by Reference
What is passed	Copy of value	Address (reference) of variable
Changes in function	Do not affect original variable	Affect original variable
Memory	Separate memory for copy	Same memory used (via pointer)
Default in C	Yes	No (must use pointers)

Passing Simple Parameter vs Passing Array in C

1. Passing Simple Parameter (Variable)

- Works as **Call by Value** in C.
- Only a **copy of the variable** is passed to the function.
- Changes made inside the function **do not affect the original variable**.

Example:

```
#include <stdio.h>
void change(int x) {
    x = 20;   // only local copy changes
}
int main() {
    int a = 10;
    change(a);
    printf("a = %d", a);   // Output: 10 (unchanged)
    return 0;
}
```

2. Passing Array

- When we pass an array, we actually pass the **address of the first element**.
- Works similar to **Call by Reference**.
- Changes inside the function **affect the original array**.

Example:

```
#include <stdio.h>
void modify(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        arr[i] = arr[i] * 2;   // modifies original array
    }
}
int main() {
    int nums[3] = {1, 2, 3};
    modify(nums, 3);
    for(int i = 0; i < 3; i++) {
        printf("%d ", nums[i]);
    }
    return 0;
}
```

Output: `2 4 6` (original array modified)

3. Key Differences

Feature	Passing Simple Parameter	Passing Array
Type	Call by Value	Call by Reference (address passed)
What is passed	Copy of variable value	Address of first element of array
Effect on original	Original variable remains same	Original array elements can change
Memory usage	Extra memory for copy	No extra copy (same memory used)
Example	`func(a);`	`func(arr, size);`