Loops in C (for, while, do-while, break, continue)

1. Theoretical Understanding

1.1 Looping Constructs in C

Looping allows repeating a block of code multiple times.

Loop Type	Entry/ Exit	When to Use
`for`	Entry	Fixed number of iterations
`while`	Entry	Condition-based iterations
`do-while`	Exit	Run at least once

1.2 Syntax

```
// for loop
for(initialization; condition; increment) {
    // code block
}

// while loop
while(condition) {
    // code block
}

// do-while loop
do {
    // code block
} while(condition);
```

1.3 break vs continue

Keyword	What it does
`break`	Terminates the loop immediately
`continue`	Skips current iteration and moves to next loop

2. Key Differences: for vs while vs do-while

Feature	`for`	`while`	`do-while`
Entry/Exit	Entry controlled	Entry controlled	Exit controlled
Use case	Counted loops	Conditional loops	Run at least once
Terminates on	Condition false	Condition false	Condition false (after 1 run)

3. Code Examples

3.1 Simple for Loop

```
for (int i = 1; i <= 5; i++) {
    printf("%d ", i);
}
// Output: 1 2 3 4 5
```

3.2 While Loop

```
int i = 1;
while (i <= 5) {
    printf("%d ", i);
    i++;
}
// Output: 1 2 3 4 5
```

3.3 Do-While Loop

```
int i = 1;
do {
    printf("%d ", i);
    i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

3.4 break Example

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    printf("%d ", i);
}
// Output: 1 2
```

3.5 continue Example

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    printf("%d ", i);
}
// Output: 1 2 4 5
```

4. Dry Run Table (Trainer Activity)

```
For this code:
int i = 1;
while (i < 5) {
    if (i == 3) break;
    printf("%d ", i);
    i++;
}
```

Iteration	i	Condition	Inside `if`?	Printed	New i
1	1	TRUE	FALSE	1	2
2	2	TRUE	FALSE	2	3
3	3	TRUE	true (break)		—

5. Practical Problems (In-Class)

Beginner Level:

1. Print even numbers from 1 to 50.
2. Count digits in a number.
3. Print sum of first n numbers.

Intermediate:

4. Reverse a number using while loop.
5. Find factorial of a number.
6. Print multiplication table of a number.

MCQ:

What is the output of this?

```
for (int i = 0; i < 5; i++) {
    if (i == 2)
        break;
    printf("%d ", i);
}
```

Output: 0 1

6. Common Mistakes & Tips

Mistake	Correction
`while (i < 5);` (semicolon ends loop)	Remove `;` — it makes it an empty loop
Forgetting `i++` → Infinite loop	Always update loop variable
Confusing `break` vs `continue`	Dry-run helps

7. Homework Assignments

Topic-Aligned LeetCode:

- FizzBuzz
- Count and Say

Write Code:

1. Print numbers from 1 to n without using semicolon (;).
2. Reverse digits of a number using while.

Nested Loops + Pattern Printing

1. Theory: Nested Loops

What is a Nested Loop?

A loop inside another loop.

```
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++) {
      printf("*");
   }
   printf("\n");
}
```

Real-Life Analogy:

- Outer loop = number of rows
- Inner loop = characters per row

How it works:

Each time the **outer loop runs once**, the **inner loop runs completely**.

2. Common Patterns

```
|Right Angle Triangle |
*
* *
* * *
| Inverted triangle |
* * *
* *
*
```

```
| Number triangle |
1
1 2
1 2 3
| Pyramid |
     *
   * * *
 * * * * *
```

3. Code Examples

Pattern 1: Right-Angled Triangle of Stars

```
for (int i = 1; i <= 5; i++) {
   for (int j = 1; j <= i; j++) {
      printf("* ");
   }
   printf("\n");
}
```

Pattern 2: Number Triangle

```
for (int i = 1; i <= 5; i++) {
   for (int j = 1; j <= i; j++) {
      printf("%d ", j);
   }
   printf("\n");
}
```

Pattern 3: Alphabet Triangle

```
char ch = 'A';
for (int i = 1; i <= 4; i++) {
   for (int j = 1; j <= i; j++) {
      printf("%c ", ch);
      ch++;
   }
   printf("\n");
}
```

Pattern 4: Inverted Triangle

```
for (int i = 5; i >= 1; i--) {
   for (int j = 1; j <= i; j++) {
      printf("* ");
   }
   printf("\n");
}
```

4. Dry Run Activity (Trainer Use)

```
for (int i = 1; i <= 3; i++) {
   for (int j = 1; j <= i; j++) {
      printf("%d", j);
   }
   printf("\n");
}
```

i	j (inner loop)	Printed Output
1	1	1
2	1 2	12
3	1 2 3	123

5. Practice Problems (Live Coding / Pair Programming)

Beginner:

1. Print a triangle with n rows using stars.
2. Print a triangle using numbers from 1 to n.

Intermediate:

3. Right-align a triangle (space padding).
4. Print a triangle with characters A to Z.

Output MCQ:

What is the output?

```
for (int i = 1; i <= 2; i++) {
   for (int j = 1; j <= 3; j++) {
      if (i == j) break;
      printf("* ");
   }
}
Output: * * *
```

6. Smart Tricks & Observations

Concept	Tip
Triangle Shape	Outer loop = number of rows
Columns logic	Inner loop = pattern elements
Right-align patterns	Use space loop first
Character patterns	Use `'A' + i` or `'a' + j` for alphabets

7. Homework

Write Programs:

1. Print Floyd's Triangle
   1
2. 2 3
3. 4 5 6
4. 7 8 9 10
5.
6. Pyramid of stars (centered with spaces).
7. Inverted alphabet triangle:
   D C B A
8. C B A
9. B A
10. A
11.

LeetCode-style:

- Print Patterns *(community pattern discussion)*
- Reverse Triangle logic using loops

Pointers in C

1. Theory: What are Pointers?

A Pointer:

A variable that stores the **address** of another variable.

```
int a = 10;
int *ptr = &a;
```

Term	Meaning
`int *ptr`	Declares pointer to an integer
`&a`	Address of `a`
`*ptr`	Dereference pointer to access value at that address

Pointer Analogy:

Pointer = Door to a room (variable). You don’t move the room, you move through the door.

2. Code Examples

2.1 Basic Pointer Usage

```
int a = 5;
int *p = &a;

printf("%d\n", a);      // 5
printf("%p\n", &a);     // address of a
printf("%p\n", p);      // same address
```

```
printf("%d\n", *p);   // 5 (dereferencing)
```

2.2 Changing Value via Pointer

```
int a = 10;
int *p = &a;

*p = 20;
printf("%d\n", a);  // Output: 20
```

3. Types of Pointers (Conceptual Understanding)

Type	Example
Null Pointer	`int *p = NULL;`
Dangling Pointer	Pointer to freed memory
Wild Pointer	Declared but not initialized
Void Pointer	`void *p;` (generic pointer)
Pointer to Pointer	`int **p2 = &p1;`

4. Pointer with Array

```
int arr[3] = {10, 20, 30};
int *p = arr;   // or &arr[0]

printf("%d\n", *(p + 1)); // Output: 20
```

Expression	Meaning
`arr`	base address
`p + i`	address of ith element
`*(p + i)`	value at that address

5. Dry Run (Trainer Activity)

```
int x = 5;
int *p = &x;
int **q = &p;

printf("%d\n", **q);
```

Variable	Value	Meaning
`x`	5	int
`p`	&x	pointer to x
`q`	&p	pointer to pointer
`**q`	5	dereferencing twice

6. Practical Problems (Live Coding)

Beginner:

1. Print address and value of a variable using a pointer.
2. Modify value using pointer.

Intermediate:

3. Print array elements using pointers.
4. Count number of even numbers in array using pointer iteration.

Advanced ():

5. Reverse an array using pointers.
6. Swap two variables using pointers.
7. Find the largest number in array using pointer logic.

7. Output MCQs

❓ Q1:

```
int a = 10;
int *p = &a;
*p = *p + 5;
printf("%d\n", a);
Output: 15
```

❓ **Q2:**

```
int arr[] = {1, 2, 3, 4};
int *p = arr;
printf("%d", *(p + 2));
Output: 3
```

8. Pointer Pitfalls

Mistake	Fix
Using uninitialized pointer (wild)	Always initialize
Forgetting `*` for dereference	Use `*ptr` for value
Confusing `*ptr` and `&ptr`	Practice dry-run
Segmentation Fault	Happens with NULL/ dangling

9. Homework

Coding Practice:

1. Write a program to print array in reverse using pointers.
2. Write a function to swap two numbers using pointers.
3. Use pointer to count number of vowels in a string.

LeetCode-style:

- Reverse String
- Swap Numbers *(Pointer logic from linked lists can be simplified for arrays)*

Pointers in Details

1. Pointer Declaration and Initialization

```
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x;

    printf("Value of x: %d\n", x);          // 10
    printf("Address of x: %p\n", &x);       // Address
    printf("Value at ptr: %d\n", *ptr);     // 10
    printf("Pointer holds: %p\n", ptr);     // Address of x
    return 0;
}
```

Use: Access variables indirectly and modify them using pointers.

2. Pointer Arithmetic

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30};
    int *ptr = arr;

    printf("%d\n", *ptr);      // 10
    ptr++;
    printf("%d\n", *ptr);      // 20
    ptr += 1;
    printf("%d\n", *ptr);      // 30
    return 0;
}
```

Use: Efficient array traversal.

3. Pointers with Arrays

```
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    int *ptr = arr;
    for (int i = 0; i < 3; i++) {
        printf("%d ", *(ptr + i)); // 1 2 3
    }
    return 0;
}
```

Use: Treat array name as a pointer.

4. Pointer to Pointer

```
#include <stdio.h>

int main() {
    int x = 5;
    int *ptr = &x;
    int **pp = &ptr;

    printf("Value: %d\n", x);       // 5
    printf("Via *ptr: %d\n", *ptr); // 5
    printf("Via **pp: %d\n", **pp); // 5
    return 0;
}
```

Use: Dynamic 2D arrays, matrix, complex references.

5. Pointers as Function Arguments (Swap Example)

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    printf("x: %d, y: %d\n", x, y); // x: 20, y: 10
    return 0;
}
```

Use: Modify variables in-place using reference.

6. Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr = malloc(3 * sizeof(int));
    arr[0] = 1; arr[1] = 2; arr[2] = 3;

    for (int i = 0; i < 3; i++) {
        printf("%d ", arr[i]);  // 1 2 3
    }

    free(arr); // Don't forget!
    return 0;
}
```

Use: Allocate memory at runtime, resize later.

7. Dangling Pointer Example

```
#include <stdio.h>
#include <stdlib.h>

int *create() {
    int *ptr = malloc(sizeof(int));
    *ptr = 100;
    return ptr;
}

int main() {
    int *data = create();
    printf("%d\n", *data); // 100

    free(data);
    // printf("%d\n", *data); // Dangling pointer, accessing after free
    return 0;
}
```

Accessing freed memory causes **undefined behavior**.

8. Array of Pointers (Strings)

```
#include <stdio.h>
```

```
int main() {
    char *names[] = {"Alice", "Bob", "Charlie"};

    for (int i = 0; i < 3; i++) {
        printf("%s\n", names[i]);
    }
    return 0;
}
```

Use: Manage list of strings, function names, etc.

9. Pointer Expression Tracing

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;
    int *p = &x, *q = &y;
    int **r = &p;

    printf("%d\n", **r);  // 10
    *r = q;
    printf("%d\n", **r);  // 20
    return 0;
}
```

/MCQ favorite — test understanding of pointer relationships.

10. Structs and Pointers

```
#include <stdio.h>

struct Student {
    int id;
    char name[20];
};

int main() {
    struct Student s = {101, "Arun"};
    struct Student *ptr = &s;

    printf("ID: %d\n", ptr->id);     // 101
    printf("Name: %s\n", ptr->name); // Arun
    return 0;
}
```

Arrays in C (Declaration, Initialization, Passing to Function, Returning)

1. Theory: Arrays Basics

What is an Array?

A collection of elements of the **same type** stored **contiguously** in memory.

```
int arr[5];              // Declaration
int arr[] = {1, 2, 3};   // Declaration + Initialization
```

Concept	Meaning
`arr[0]`	Access 1st element
`arr[i]`	ith element
`*(arr + i)`	Pointer-style access

Key Points:

- Indexing starts from 0
- Stored in **contiguous memory**
- Name of array = base address (arr == &arr[0])

2. Dry Run (Memory Representation)

int arr[3] = {10, 20, 30};

Index	Value	Address (ex.)
0	10	1000
1	20	1004
2	30	1008

3. Practical Code Examples

Declare and Access Elements

```
int a[3] = {1, 2, 3};
printf("%d\n", a[1]);  // Output: 2
```

Iterate Array using Loop

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

Initialize with Loop

```
int a[5];
for (int i = 0; i < 5; i++) {
    a[i] = i + 1;
}
```

4. Passing Array to Function

✉️ Pass Array as Argument

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

🧪 Call:

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

Note: Passing array = passing address (no copy).

5. Returning Array from Function

C doesn't allow returning whole arrays. We return **pointer** to array or use **static array** inside function.

Using Static Array

```
int* getArray() {
    static int arr[3] = {1, 2, 3};
    return arr;
}
```

Warning:

Returning local arrays (non-static) causes **undefined behavior**.

6. Practice Problems

Beginner

1. Write a program to take n integers from user and print sum.
2. Find maximum and minimum in array.
3. Count even and odd numbers in array.

Intermediate

4. Search for an element in array (linear search).
5. Reverse array using a function.
6. Calculate frequency of each element.

Output-based MCQs

❓ **Q1:**

```
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", *(arr + 3));
Output: 4
```

❓ **Q2:**

```
void change(int arr[]) {
    arr[0] = 100;
}
int arr[] = {1,2,3};
```

```
change(arr);
printf("%d", arr[0]);
Output: 100
```

(Arrays are passed by reference)

7. Array with Functions Summary

Task	Function Style
Print array	`void print(int[], int)`
Modify array	`void update(int[])`
Return array	`int* getArray()` with `static`

8. Homework

Concept Practice

1. Implement linear search function.
2. Create a function to return the sum of all array elements.
3. Return largest and second largest in an array.

LeetCode-style:

- Find Maximum Number
- Reverse Array
- Find Second Largest Element

Arrays in details

1. Declaration and Initialization

```
#include <stdio.h>

int main() {
   int arr[5];              // Declaration only
   int brr[5] = {1, 2, 3};    // Partial init → brr[3], brr[4] = 0
   int crr[] = {10, 20, 30};  // Size deduced

   printf("%d\n", crr[1]);    // Output: 20
   return 0;
}
```

Key: Default uninitialized values are garbage unless explicitly initialized.

2. Traversing an Array

```
#include <stdio.h>

int main() {
   int arr[5] = {10, 20, 30, 40, 50};
   for (int i = 0; i < 5; i++) {
      printf("%d ", arr[i]); // Output: 10 20 30 40 50
   }
   return 0;
}
```

Trainer Tip: Stress index range (0 to n-1), show out-of-bound errors.

3. Passing Arrays to Functions

```
#include <stdio.h>

void printArray(int a[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%d ", a[i]);
   }
}

int main() {
   int arr[3] = {1, 2, 3};
   printArray(arr, 3);  // Output: 1 2 3
   return 0;
}
```

Important: Arrays are passed by **reference** (i.e., pointer).

4. Array as Pointers

```
#include <stdio.h>

int main() {
   int arr[3] = {100, 200, 300};
   int *ptr = arr;

   for (int i = 0; i < 3; i++) {
      printf("%d ", *(ptr + i));  // 100 200 300
   }
   return 0;
}
```

Shows pointer–array equivalence. Ask: What is *(arr + 2)?

5. Reverse an Array (In-place)

```
#include <stdio.h>

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int n = 5;

   for (int i = 0; i < n/2; i++) {
      int temp = arr[i];
      arr[i] = arr[n-1-i];
      arr[n-1-i] = temp;
   }

   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);  // 5 4 3 2 1
   }
   return 0;
}
```

6. Find Second Largest Element

```
#include <stdio.h>

int main() {
   int arr[] = {10, 50, 30, 20, 40};
   int first = arr[0], second = -1;

   for (int i = 1; i < 5; i++) {
      if (arr[i] > first) {
         second = first;
         first = arr[i];
      } else if (arr[i] > second && arr[i] != first) {
         second = arr[i];
      }
   }

   printf("Second largest: %d\n", second); // Output: 40
   return 0;
}
```

Most asked pattern!

7. Check Array Palindrome

```
#include <stdio.h>

int main() {
   int arr[] = {1, 2, 3, 2, 1}, isPal = 1;

   for (int i = 0; i < 5/2; i++) {
      if (arr[i] != arr[4-i]) {
         isPal = 0;
         break;
      }
   }

   if (isPal)
      printf("Palindrome\n");
   else
      printf("Not a palindrome\n");
   return 0;
}
```

8. Count Frequency of Each Element

```
#include <stdio.h>

int main() {
   int arr[] = {1, 2, 2, 3, 1};
   int freq[100] = {0};

   for (int i = 0; i < 5; i++) {
      freq[arr[i]]++;
   }

   for (int i = 0; i < 5; i++) {
      if (freq[arr[i]] != 0) {
         printf("%d → %d times\n", arr[i], freq[arr[i]]);
         freq[arr[i]] = 0;  // avoid duplicate print
      }
   }
   return 0;
}
```

9. Merge Two Sorted Arrays

```
#include <stdio.h>

int main() {
   int a[] = {1, 3, 5}, b[] = {2, 4, 6};
   int c[6], i = 0, j = 0, k = 0;

   while (i < 3 && j < 3) {
      c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
   }
   while (i < 3) c[k++] = a[i++];
   while (j < 3) c[k++] = b[j++];

   for (int x = 0; x < 6; x++) printf("%d ", c[x]);
   return 0;
}
```

10. Pass Array and Return from Function

```
#include <stdio.h>
int* getSquares(int arr[], int size) {
   static int result[10];
   for (int i = 0; i < size; i++) {
       result[i] = arr[i] * arr[i];
   }
   return result;
}

int main() {
   int input[] = {1, 2, 3}, *output = getSquares(input, 3);
   for (int i = 0; i < 3; i++) {
       printf("%d ", output[i]);  // 1 4 9
   }
   return 0;
}
```

🛑 Static used here so return is safe.

Dynamic Memory Management in C

1. Theory: Why Dynamic Memory?

Static Memory	Dynamic Memory
Size decided at compile time	Size decided at runtime
Uses stack	Uses heap
Memory auto-managed	Programmer must allocate & free
Fast but limited	Flexible but error-prone

Common Scenarios:

- Need array size at runtime.
- Memory optimization in real-time systems.
- Handling unknown or huge data inputs.

2. malloc()

malloc (Memory Allocation):

Allocates **uninitialized** memory block of specified bytes.

```
int *ptr = (int *)malloc(5 * sizeof(int));
```

Component	Description
malloc()	Allocates memory
sizeof(int)	Gets size of type
(int *)	Typecasting (C-style)
ptr	Points to start of block

Note:

- Memory may contain garbage values.
- Always check if allocation succeeded:

```
if (ptr == NULL) {
    printf("Memory not allocated");
}
```

3. calloc()

Allocates **zero-initialized** memory.

```
int *ptr = (int *)calloc(5, sizeof(int));
```

Parameter	Meaning
5	Number of blocks
`sizeof(int)`	Size per block
Memory content	All zero

4. realloc()

Reallocates memory to increase or decrease previously allocated size.

```
ptr = (int *)realloc(ptr, 10 * sizeof(int));
```

Use Case:

- Need to resize array during runtime.

5. free()

Used to **release** dynamically allocated memory back to heap.

free(ptr);

Warning	**Avoid using pointer after `free()` – becomes dangling**

6. Practical Examples

malloc()

```
int *arr = (int *)malloc(3 * sizeof(int));
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

calloc()

```
int *arr = (int *)calloc(3, sizeof(int));  // All values: 0
```

realloc()

```
arr = (int *)realloc(arr, 5 * sizeof(int));
arr[3] = 4; arr[4] = 5;
```

free()

```
free(arr);  // Always free when done
```

7. Trainer Dry Run Example

```
int *ptr = (int *)malloc(2 * sizeof(int));
ptr[0] = 10; ptr[1] = 20;
ptr = (int *)realloc(ptr, 3 * sizeof(int));
ptr[2] = 30;
```

Trace Memory:

- malloc: 2 ints → 8 bytes
- realloc: resized to 12 bytes
- Values: 10, 20, 30

8. Practical Problems

Beginner

1. Allocate array using malloc, input values and print sum.
2. Compare malloc vs calloc by printing values after allocation.

Intermediate

3. Take dynamic input size n, use malloc and print all even elements.

4. Reallocate array when size increases and fill additional values.

Advanced ()

5. Store n employee records (id, name, salary) using struct + malloc.

6. Resize employee array on demand using realloc.

7. Free all memory safely before program ends.

9. MCQ & Output Tracing

❓ Q1:

```
int *p = (int *)calloc(3, sizeof(int));
printf("%d", p[2]);
```

Output: 0

❓ Q2:

```
int *a = (int *)malloc(2 * sizeof(int));
a[0] = 5; a[1] = 10;
a = (int *)realloc(a, 3 * sizeof(int));
a[2] = 15;
printf("%d", a[2]);
```

Output: 15

10. Homework

Concepts

1. Use malloc to create a dynamic array, input size and values, and print max and min.

2. Use calloc to allocate a matrix (2D) and initialize all elements to 0.

3. Use realloc to increase array size and compute new average.

LeetCode-style (Pointer + Memory Inspired):

- Dynamic Array Simulation – Custom Implementation *(Design logic)*
- Two Sum *(Can be enhanced with realloc + malloc combo)*
- Merge Sorted Arrays *(Needs dynamic resizing if not in-place)*

String in C

In C, **a string is an array of characters** that ends with a special null character '\0' to indicate the end of the string.

```
char name[] = "";  // Actually stored as: 'Z' 'o' 'h' 'o' '\0'
```

1. String Declarations

A. Using character array:

```
char name[10] = "";  // Can modify content
```

B. Using pointer to string literal:

```
char *name = "";     // Stored in read-only memory (don't modify)
```

Trying name[0] = 'z'; in the second case → segmentation fault

2. Input and Output

Using scanf (stops at whitespace)

```
char name[50];
scanf("%s", name);
```

Using fgets (recommended, reads full line)

```
fgets(name, sizeof(name), stdin);
```

3. Memory Allocation

Method	Memory Type	Notes
`char str[]`	Stack	Auto size, fixed, mutable
`char *str`	Read-only	Literal, cannot modify
`malloc()`	Heap	Use for dynamic strings

```
char *str = malloc(50 * sizeof(char));
strcpy(str, "Hello");
```

Always use free(str) after use.

4. String Functions (#include <string.h>)

Function	Description
`strlen(s)`	Length (no null terminator)
`strcpy(dest, src)`	Copy string
`strncpy(dest, src, n)`	Copy first n characters
`strcmp(s1, s2)`	Compare strings (lexical)
`strcat(s1, s2)`	Concatenate
`strchr(s, c)`	First occurrence of char
`strstr(s, sub)`	Find substring
`strrev(s)`	Reverse string (non-standard)

5. String Traversal Example

```
void printChars(char *str) {
    while (*str != '\0') {
        printf("%c ", *str);
        str++;
    }
}
```

6. Sample Programs

A. Reverse a String

```
void reverse(char *s) {
    int l = 0, r = strlen(s) - 1;
    while (l < r) {
        char temp = s[l];
        s[l] = s[r];
        s[r] = temp;
        l++; r--;
    }
}
```

B. Check for Palindrome

```
int isPalindrome(char *s) {
    int l = 0, r = strlen(s) - 1;
    while (l < r) {
        if (s[l++] != s[r--])
            return 0;
    }
    return 1;
}
```

C. Custom strlen Implementation

```
int my_strlen(const char *str) {
    int len = 0;
    while (*str++) len++;
    return len;
}
```

7. String vs Character Array vs Pointer

```
char s1[] = "";     // Stored in stack, modifiable
char *s2 = "";      // Stored in text segment, read-only
```

Aspect	`char s1[]`	`char *s2`
Storage	Stack	Read-only section
Modifiable	Yes	No
Reallocation	No (use malloc)	If dynamic allocated

8. Advanced Concepts

⚠️ Null Terminator Importance

If '\0' is missing, functions like strlen, printf, strcpy may read beyond buffer causing **garbage output or crash**.

🧠 Example: Dangerous Operation

```
char str[3] = {'Z', 'o', 'h'};  // No '\0'
printf("%s", str);  // Undefined behavior
```

String MCQ

```
char str[] = "abc";
str[1] = 'z';
printf("%s", str);  // Output: azc
```

9. Favorite String Challenges

1. Reverse string without using library functions
2. Count vowels/consonants
3. Remove duplicate characters
4. Check anagram
5. Compress string: "aaabb" → "a3b2"
6. Frequency count
7. Substring count without using strstr
8. Tokenize string with strtok()

10. Coding Tip: strtok() Example

```
char str[] = " loves code";
char *token = strtok(str, " ");
while (token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL, " ");
}
```

Structures & Unions in C

1. Theory: Structures in C

What is a Structure?

A user-defined data type that groups variables of **different types**.

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

Declaration and Initialization

struct Student s1 = {101, "Arun", 89.5};

2. Accessing Members

```
printf("%d", s1.id);
scanf("%f", &s1.marks);
```

Using Pointer:

```
struct Student *p = &s1;
printf("%s", p->name);
```

Arrow operator (->) is used with structure pointers.

3. Passing Structures to Functions

By Value:

```
void display(struct Student s) {
    printf("%d %s", s.id, s.name);
}
```

By Reference (Efficient):

```
void update(struct Student *s) {
    s->marks = 95.5;
}
```

Preferred in large programs

4. Nested Structures

```
struct Address {
    char city[20];
    int pincode;
};

struct Customer {
    int id;
    struct Address addr;
};
printf("%s", cust.addr.city);
```

Real-time application: Customer → Address (Aggregation)

5. Union in C

Union is similar to structure but shares the same memory for all members.

```
union Data {
    int i;
    float f;
    char ch;
};
```

- Only **one member** is stored at a time.
- Memory = size of **largest member**.

Structure vs Union (Memory Layout)

Feature	Structure	Union
Memory	Sum of all members	Max of one member
Access	All at once	One at a time
Use-case	Group data	Memory optimized

6. Practical Problems

Beginner:

1. Define a struct Employee with id, name, salary. Input & print details.
2. Create a function to update salary by 10% (pass by reference).

Intermediate:

3. Define struct Customer with embedded struct Address. Take input & display full profile.
4. Pass an array of struct Student to function and print class average.

:

5. Use malloc() to dynamically allocate an array of struct Product and print total stock value.
6. Create a billing system using nested struct: Customer -> Address, Order -> Product.

7. MCQ & Output Tracing ()

❓ Q1:

```
struct Test {
    int x;
    char y;
    float z;
};
printf("%lu", sizeof(struct Test));
```

Output: Usually 12 or 16 depending on padding.

❓ Q2:

```
union Test {
    int x;
    char y;
    float z;
};
printf("%lu", sizeof(union Test));
```

Output: 4 (if float is largest)

❓ Q3:

```
struct Test {
    int a;
    struct {
        int x, y;
    } point;
};
```

Nested struct access: t.point.x

8. Homework

Struct Design Practice:

1. Design a Library system struct: Book, Author, Publisher.
2. Write a program to take 3 employee records and print the one with highest salary.
3. Use nested struct for Student and Department, and input full details.

LeetCode-style Analogies (for practice logic):

- Design a Parking System
- Employee Importance
- Design a Movie Ticket Booking System