String in C Language

1. Introduction

- In C language, a **string** is a sequence of characters that ends with a **null character (`\0`)**.
- Strings are widely used to store and manipulate text.
- In C, there is **no special string data type**. Strings are represented using **character arrays**.

Example:

```
char str[10] = "Hello";
```

Here, memory stores it as:

```
H   e   l   l   o   \0
```

2. Declaration of Strings

Strings can be declared in two ways:

1. **Using character array**

```
char str[6] = {'H','e','l','l','o','\0'};
```

2. **Using string literal** (easy method)

```
char str[] = "Hello";
```

3. Input and Output of Strings

Output (`printf`)

```
char name[] = "Neeraj";
printf("%s", name);   // Output: Neeraj
```

Input (`scanf`)

```
char name[20];
scanf("%s", name);    // Input: Neeraj
printf("Hello %s", name);
```

Note: `scanf("%s", name)` stops input at the first space.

For full line input (with spaces), use `gets()` or `fgets()`.

```
char sentence[50];
fgets(sentence, sizeof(sentence), stdin);
printf("%s", sentence);
```

4. String Functions (from <string.h>)

C provides many built-in functions for string handling:

1. **`strlen(str)`** – returns length of string (excluding `\0`).

```
printf("%d", strlen("Hello")); // 5
```

2. **`strcpy(dest, src)`** – copies one string into another.

```
char str1[20], str2[20] = "World";
strcpy(str1, str2);  // str1 = "World"
```

3. **`strcat(str1, str2)`** – concatenates two strings.

```
char a[20] = "Hello ", b[] = "World";
strcat(a, b);  // a = "Hello World"
```

4. **`strcmp(str1, str2)`** – compares two strings.

    - Returns `0` if equal.
    - Returns `>0` if str1 > str2.
    - Returns `<0` if str1 < str2.

```
strcmp("abc", "abc"); // 0
strcmp("abc", "abd"); // -1
```

5. **`strupr(str)`** (compiler dependent) – converts to uppercase.
6. **`strlwr(str)`** (compiler dependent) – converts to lowercase.
7. **`strrev(str)`** (compiler dependent) – reverses string.

Structures in C Language

- A **structure (`struct`)** in C is a user-defined data type.
- It allows grouping of **different data types** (heterogeneous data) under one name.
- Useful when you want to store related information together (like details of a student, employee, book, etc.).

Example: Student Structure

```
#include <stdio.h>

// Defining structure
struct Student {
    char name[50];   // string for name
    int age;         // integer for age
    float marks;     // float for marks
};

int main() {
    // Creating structure variable
    struct Student s1;

    // Assigning values
    printf("Enter name: ");
    scanf("%s", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);

    printf("Enter marks: ");
```

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

    // Displaying values
    printf("\n--- Student Details ---\n");
    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

Explanation

- `struct Student` → defines a new data type with members `name`, `age`, `marks`.
- `s1` → variable of type `struct Student`.
- Access members with **dot operator (`.`)** → `s1.name`, `s1.age`, `s1.marks`.

Key Points about Structures

1. Can hold **different data types** in one unit.
2. **Dot operator** (`.`) is used to access members.
3. Multiple variables of a structure type can be created.
4. Structures can also be used with **arrays**, **pointers**, and **functions**.

`typedef` in C Language

- `typedef` in C is a keyword used to **create a new name (alias)** for an existing data type.
- It makes code **shorter, cleaner, and more readable**.
- Does **not create a new type**, only a **nickname** for an existing type.

Syntax

```
typedef existing_datatype new_name;
```

Simple Example

```
#include <stdio.h>

typedef unsigned int uint;   // creating alias "uint" for unsigned int

int main() {
    uint age = 25;   // same as "unsigned int age = 25;"
    printf("Age = %u", age);
    return 0;
}
```

Output:

```
Age = 25
```

Using `typedef` with `struct`

Normally:

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

struct Student s1;   // must write "struct" every time
```

With `typedef`:

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

int main() {
    Student s1;   // now no need to write "struct Student"
    return 0;
}
```

Using `typedef` with Pointers

```
typedef int* IntPtr;

int main() {
    IntPtr p1, p2;   // both are int pointers
    int a = 10, b = 20;
    p1 = &a;
    p2 = &b;
    printf("%d %d", *p1, *p2);
    return 0;
}
```

Key Points

1. `typedef` is used to **rename types** for convenience.
2. Helps make complex declarations (like structures and pointers) simpler.
3. Commonly used with:
    - `unsigned int` → shorter alias (`uint`)
    - `struct` → to avoid writing `struct` keyword repeatedly
    - `pointers` → for cleaner syntax

Union in C Language

- A **union** in C is a user-defined data type like a structure.
- It can store **different data types** in the same memory location.
- **Key difference from structure**:
    - In **structure**, each member has **separate memory**.
    - In **union**, all members **share the same memory**, so only one member can hold a value at a time.

Example: Student Union

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

```
// Defining union
union Student {
    char name[50];
    int age;
    float marks;
};

int main() {
    union Student s1;

    // Assigning values one by one
    s1.age = 20;
    printf("Age = %d\n", s1.age);

    s1.marks = 88.5;
    printf("Marks = %.2f\n", s1.marks);

    // Now accessing age again will give garbage
    printf("Age after assigning marks = %d\n", s1.age);

    return 0;
}
```

Output (Approximate)

```
Age = 20
Marks = 88.50
Age after assigning marks = Garbage value
```

Memory in Union

- In **structure** → total memory = sum of all members.
- In **union** → total memory = size of **largest member**.

Example:

```
struct StudentStruct {
    char name[50];   // 50 bytes
    int age;         // 4 bytes
    float marks;     // 4 bytes
};
// Total = 50 + 4 + 4 = 58 bytes (approx alignment may increase)

union StudentUnion {
    char name[50];   // 50 bytes
    int age;         // 4 bytes
    float marks;     // 4 bytes
};
// Total = max(50, 4, 4) = 50 bytes
```

Key Points about Union

1. **One memory block shared** by all members.
2. At any point, only the **last assigned member** holds a meaningful value.
3. **Memory efficient** compared to structures.
4. Useful in cases like:
    - Embedded systems (to save memory).

- Storing different data types in the same location (e.g., interpreting network packets).

Enum (Enumeration) in C Language

What is `enum`?

- `enum` (short for *enumeration*) is a **user-defined data type** in C.
- It is used to assign **names to a set of integer constants**, which makes the code more readable.
- By default, the first name gets value **0**, the next **1**, and so on (unless values are specified manually).

Example 1: Days of the Week

```
#include <stdio.h>

enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY };

int main() {
    enum Weekday today;
    today = WEDNESDAY;

    printf("Day number: %d\n", today);   // Output: 2 (since MONDAY=0,
TUESDAY=1, WEDNESDAY=2)

    return 0;
}
```

Example 2: Custom Values

```
#include <stdio.h>

enum Status { SUCCESS = 1, FAILURE = -1, PENDING = 0 };

int main() {
    enum Status s1 = SUCCESS;
    enum Status s2 = FAILURE;

    printf("s1 = %d\n", s1);  // Output: 1
    printf("s2 = %d\n", s2);  // Output: -1

    return 0;
}
```

Key Points about `enum`

1. **Default values** start from 0 and increase by 1.
   Example: `{A, B, C}` → A=0, B=1, C=2.
2. We can **manually assign values**.
3. Enums improve **readability** (using names instead of numbers).
4. Internally, enum variables are stored as **integers**.
5. Enum constants are **compile-time constants** (like `#define`).

Difference Between `enum` and `#define`

Feature	`enum`	`#define`
Type Safety	Stronger (treated as int)	Just text replacement
Debugging	Easier (can print values)	Harder to debug
Scope	Limited to block/file scope	Global, no scope control
Usage	Group of related constants	Single constant or macro

Summary:
`enum` is a **user-defined data type** in C that assigns names to integral constants, making programs **more readable and maintainable**.

Static Variables in C Language

Definition

A **static variable** in C is a variable that retains its value **between multiple function calls**. It is initialized only once and its lifetime is throughout the execution of the program.

Key Points:

1. Declared using the keyword **`static`**.
2. **Scope:** Limited to the block/function where it is declared.
3. **Lifetime:** Entire program execution (not destroyed after function ends).
4. **Default value:** `0` (if not explicitly initialized).
5. **Storage:** Stored in the **Data Segment** (not in stack like auto variables).

Syntax

```
static data_type variable_name = value;
```

Example 1: Static inside a function

```
#include <stdio.h>

void demo() {
    static int count = 0; // initialized only once
    count++;
    printf("Count = %d\n", count);
}

int main() {
    demo();
    demo();
    demo();
    return 0;
}
```

Output

```
Count = 1
Count = 2
```

```
Count = 3
```

Memory Layout in C Language

When a C program is compiled and executed, its memory is divided into different segments. Each segment has a specific purpose:

1. Text Segment (Code Segment)

- Contains the **compiled machine instructions** (program code).
- **Read-only** → prevents modification of code at runtime.
- Example: functions like `main()`, `printf()`, etc.

2. Data Segment

This is divided into two parts:

- **a) Initialized Data Segment**
  - Stores **global** and **static** variables that are explicitly initialized.
  - Example:

```
int x = 10;       // stored in initialized data
static int y = 20; // stored in initialized data
```

- **b) Uninitialized Data Segment (BSS Segment)**
  - Stores **global** and **static** variables that are not initialized.
  - Default value = **0**.
  - Example:

```
int a;         // stored in BSS
static int b;  // stored in BSS
```

3. Stack Segment

- Used for **function calls** and **local variables**.
- Stores function parameters, return addresses, and local variables.
- Follows **LIFO (Last In, First Out)** principle.
- Each function call creates a **stack frame**, which is removed after the function returns.
- Grows **downward** (towards lower memory addresses).
- Example:

```
void func() {
    int x = 5;   // stored in stack
}
```

4. Heap Segment

- Used for **dynamic memory allocation** during runtime.
- Managed using functions like `malloc()`, `calloc()`, `realloc()`, and `free()`.
- Grows **upward** (towards higher memory addresses).
- Example:
- `int *p = (int*)malloc(5 * sizeof(int));  // stored in heap`

Dynamic Memory Allocation in C

Dynamic Memory Allocation (DMA) in C allows programmers to allocate memory **at runtime** instead of compile time.
This provides flexibility when the size of data is not known in advance.
The allocated memory comes from the **Heap Segment**.

Why Use Dynamic Memory Allocation?

- Memory size can be decided **at runtime**.
- Efficient use of memory → allocate when needed, release when done.
- Useful for working with **data structures** like linked lists, trees, graphs, etc.

Functions for Dynamic Memory Allocation

All these functions are declared in **`<stdlib.h>`**:

1. malloc()

- Stands for **Memory Allocation**.
- Allocates a **block of memory** of given size (in bytes).
- Returns a **pointer** to the first byte of allocated memory.
- Memory is **uninitialized** (contains garbage values).

Syntax:

```
ptr = (castType*) malloc(size_in_bytes);
```

Example:

```
int *p = (int*) malloc(5 * sizeof(int));  // allocates space for 5 integers
```

2. calloc()

- Stands for **Contiguous Allocation**.
- Allocates **multiple blocks** of memory and initializes them to **0**.
- Useful when memory must be cleared before use.

Syntax:

```
ptr = (castType*) calloc(num_elements, size_of_each);
```

Example:

```
int *p = (int*) calloc(5, sizeof(int));  // allocates space for 5 integers,
all initialized to 0
```

3. realloc()

- Stands for **Re-Allocation**.
- Used to **resize** an already allocated block of memory.
- Can increase or decrease the size without losing existing data (if possible).

Syntax:

```
ptr = (castType*) realloc(ptr, new_size_in_bytes);
```

Example:

```
p = (int*) realloc(p, 10 * sizeof(int)); // resize to store 10 integers
```

4. free()

- Used to **release memory** that was previously allocated using malloc/calloc/realloc.
- Prevents **memory leaks**.

Syntax:

```
free(ptr);
```

Storage Classes in C Language

A **storage class** in C defines the **scope, lifetime, default value, and storage location** of a variable.
There are **4 main storage classes**:

1. auto
2. register
3. static
4. extern

1. auto

- **Keyword**: `auto`
- **Default storage class** for local variables inside functions.
- **Scope**: Local to the block (function or loop).
- **Lifetime**: Created when function is called, destroyed when function ends.
- **Storage location**: **Memory (stack)**.
- **Default value**: Garbage (undefined).

Example:

```
void func() {
    auto int a = 10;   // auto storage class
    printf("%d", a);
}
```

2. register

- **Keyword**: `register`
- Suggests compiler to store the variable in a **CPU register** instead of RAM (for faster access).
- **Scope**: Local to the block.
- **Lifetime**: Till the function ends.
- **Storage location**: CPU register (if available), otherwise RAM.
- **Default value**: Garbage.
- Cannot use `&` (address operator) on register variables.

Example:

```
void func() {
    register int counter;
    for (counter = 0; counter < 5; counter++) {
        printf("%d\n", counter);
    }
}
```

3. static

- **Keyword**: `static`
- Retains its value between **multiple function calls**.
- **Scope**: Local to the block (if declared inside a function).
- **Lifetime**: Entire program execution.
- **Storage location**: **Data segment** of memory.
- **Default value**: Zero (0).

Example:

```
void func() {
    static int x = 0;   // static variable
    x++;
    printf("%d\n", x);
}

int main() {
    func();  // Output: 1
    func();  // Output: 2
    func();  // Output: 3
}
```

4. extern

- **Keyword**: `extern`
- Used to declare a **global variable** in another file or outside of the function.
- **Scope**: Global (accessible across files).
- **Lifetime**: Entire program execution.
- **Storage location**: **Data segment** of memory.
- **Default value**: Zero (0).

Example (same file):

```
int globalVar = 10;   // Global variable

void func() {
    extern int globalVar;  // reference to globalVar
    printf("%d", globalVar);
}
```

Example (multi-file program):

- `file1.c` → `int globalVar = 20;`
- `file2.c` → `extern int globalVar;`

Comparison Table of Storage Classes in C

Storage Class	Keyword	Scope	Lifetime	Storage Location	Default Value
auto	`auto`	Local (block level)	Till function ends	Stack	Garbage
register	`register`	Local (block level)	Till function ends	CPU Register/RAM	Garbage
static	`static`	Local (but persists)	Entire program	Data Segment	0
extern	`extern`	Global (all files)	Entire program	Data Segment	0