Pointers and Structs I

INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY
HYDERABAD

🧩 What is a Pointer?

A **pointer** is a variable that stores the **memory address** of another variable.

```c
int x = 10;
int *p = &x; // p stores address of x
printf("%d", *p); // prints 10
```

- `&x` gives the **address** of x.
- `*p` gives the **value stored at that address**.

🧮 Memory Visualization

Variable	Address	Value
x	1000	10
p	2000	1000

`*p` → value at address 1000 → 10

🧠 Example 1: Swapping Two Numbers

❌ Without Pointers

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

`a` and `b` are **copies** of the arguments. The swap doesn’t affect the original numbers.

✅ With Pointers

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

Example 2: Passing Arrays to Functions

Arrays are automatically passed by reference (address).

```c
void doubleArray(int *arr, int n) {
    for (int i = 0; i < n; i++)
        arr[i] *= 2;
}

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

✅ Works directly on the original array.

✅ No duplication — only base address passed.

Example 3: Returning Multiple Values

Without pointers → only one return value possible.

✅ Using pointers:

```
void compute(int a, int b, int *sum, int *prod) {
    *sum = a + b;
    *prod = a * b;
}
```

```
int s, p;
compute(5, 10, &s, &p);
printf("Sum=%d Product=%d", s, p);
```

Structs

🧩 1. Definition of a Struct

A **structure (struct)** in C is a user-defined data type that allows grouping variables of different types under a single name.

```
// Example: Defining a struct for a student
struct Student {
    int roll_no;
    char name[50];
    float marks;
};
```

Notes:

- A struct groups logically related data.
- Members can be of **different data types**.
- he keyword `struct` is used to declare a structure.

🏗️ 2. Declaring and Initializing Struct Variables

Method 1: Separate Declaration

```c
struct Student s1;
s1.roll_no = 101;
strcpy(s1.name, "Alice");
s1.marks = 89.5;
```

Method 2: Initialization at Declaration

```c
struct Student s2 = {102, "Bob", 92.0};
```

Method 3: Designated Initializers (C99 and later)

```c
struct Student s3 = {
    .roll_no = 103,
    .name = "Charlie",
    .marks = 95.2
```

🧭 3. Accessing Struct Members

You can access structure members using the **dot operator (.)**.

```c
printf("Roll No: %d\n", s1.roll_no);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);
```

💡 4. Structs with Pointers

When using pointers to structs, use the **arrow operator (->)**.

```
struct Student *ptr = &s2;
printf("Name (via pointer): %s\n", ptr->name);
ptr->marks = 93.5;
```

⚙️ 5. Array of Structs

You can create an array of structs to store multiple records.

```
struct Student class[3] = {
    {101, "Alice", 89.5},
    {102, "Bob", 92.0},
    {103, "Charlie", 95.2}
};
```

Access Example:

```
for(int i = 0; i < 3; i++) {
    printf("%d %s %.2f\n", class[i].roll_no, class[i].name, class[i].marks);
}
```

🔄 6. Passing Structs to Functions

Pass by Value

```c
void printStudent(struct Student s) {
    printf("%d %s %.2f\n", s.roll_no, s.name, s.marks);
}
```

Pass by Reference

```c
void updateMarks(struct Student *s, float newMarks) {
    s->marks = newMarks;
}
```

🧠 7. Nested Structs

Structs can contain other structs as members.

```c
struct Date {
    int day, month, year;
};

struct Student {
    int roll_no;
    char name[50];
    struct Date dob;
};
```

Access Example:

```c
struct Student s = {101, "Alice", {12, 5, 2003}};
    tf("DOB: %d/%d/%d\n", s.dob.day, s.dob.month, s.dob.year);
```