SESSION 8 - POINTERS

Chapter 1: Demystifying Pointers

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

The Analogy:

- **Variable (val):** A House.
- **Value (10):** People living in the house.
- **Address (&val):** The street address (e.g., 101 Lane).
- **Pointer (ptr):** A piece of paper with "101 Lane" written on it.

Syntax Operators

1. & (Address-of Operator): "Where are you?" (Returns address).
2. * (Dereference Operator): "Who lives there?" (Access value at address).

C

```
int num = 10;
int *ptr = &num; // ptr holds the address of num


printf("%d", num);  // Prints 10
printf("%p", ptr);  // Prints address (e.g., 0x7ffee...)
printf("%d", *ptr); // Prints 10 (Goes to address, reads value)
```

Chapter 2: Types of Pointers (Detailed)

1. Null Pointer: Points to nothing. Good practice for safety.

```
int *p = NULL;
```

2. **Void Pointer (void *):** Generic pointer. Can point to any data type, but cannot be dereferenced directly without casting.
3. Wild Pointer: Uninitialized pointer pointing to a random/garbage location. DANGEROUS.

```
int *p; // Wild!
```

4. **Dangling Pointer:** Points to memory that has been deleted or freed.
5. **Pointer to Constant vs Constant Pointer:**

Declaration	Meaning	Can change Address?	Can change Value?
const int *p	Pointer to Constant	YES	NO
int * const p	Constant Pointer	NO	YES
const int * const p	Const Ptr to Const	NO	NO

Chapter 3: Pointers with Arrays and Strings

The name of an array acts like a pointer to the first element.

C

```
int arr[] = {10, 20, 30};
int *p = arr;

// Pointer Arithmetic
printf("%d", *p);     // 10
printf("%d", *(p+1)); // 20 (Moves to next integer position)
```

Note: p+1 increases the address by sizeof(int) (4 bytes), not just 1 byte.

Activity: Swap Two Numbers using Pointers

C

```
void swap(int *a, int *b) {
   int temp = *a; // Store value at address a
   *a = *b;       // Put value at b into address a
   *b = temp;     // Put temp into address b
}

int main() {
   int x = 5, y = 10;
   swap(&x, &y); // Pass addresses
   printf("x=%d, y=%d", x, y); // x=10, y=5
   return 0;
}
```

Chapter 4: Frequently Asked Questions

Q1: Why do we need pointers?

Answer:

1. Dynamic Memory Allocation (creating memory during runtime).
2. Passing large structures to functions efficiently.
3. Implementing data structures like Linked Lists and Trees.

Q2: What is the size of a pointer?

Answer: It depends on the system architecture, not the data type it points to.

- 32-bit system: 4 bytes.
- 64-bit system: 8 bytes.

Q3: Why does my program crash with Segmentation Fault?

Answer: This usually happens when you try to dereference (*p) a Null, Wild, or Dangling pointer. You are trying to access memory that doesn't belong to you.

Q4: What is *&p?

Answer: They cancel each other out. * means "value at" and & means "address of". So *&p is just p.