Perfect! ✅ Phase 0 is fully covered. You now **think like an embedded engineer**, understand bits, registers, memory maps, timing, and deterministic behavior.

Now we move to **Phase 1 — C Foundations for Embedded Systems (Week 1–3)**. This phase is where **you start coding**, but always in the context of hardware. I'll break it **week-by-week, day-by-day**, with exercises and clear objectives.

Phase 1: C Foundations (Embedded-Oriented)

Goal: Learn C in a hardware-focused way so that every line of code maps directly to registers, memory, or hardware behavior.

Week 1 — Data Types, Variables, and Memory

Day 1–2: Understand Embedded Data Types

Unlike PC programming, in embedded:

- **Size matters:** `int` could be 16-bit or 32-bit depending on compiler
- **Exact-width types** are critical:

```
uint8_t   // unsigned 8-bit
int16_t   // signed 16-bit
uint32_t  // unsigned 32-bit
```

- **Volatile**: tells compiler **don't optimize this** — used for registers and shared memory

Exercise:

- Define all integer types and print their size with `sizeof()`
- Experiment with `volatile` variables in small programs

Example:

```
volatile uint32_t *GPIO_DATA = (uint32_t *)0x40020000; // hypothetical GPIO regis
*GPIO_DATA = 0xFF;  // sets all pins HIGH
```

Day 3–4: Variables, Scope, and Storage Classes

Understand:

- **Scope:** Local vs global vs static
- **Lifetime:** Automatic (stack) vs static (data segment)
- **Embedded importance:** Static globals often used for driver buffers; dynamic memory is avoided

Exercise:

- Write a small program to toggle an LED using static/global variables
- Track what happens in memory using a debugger or print addresses

---

Day 5–6: Pointers — The Heart of Embedded C

Pointers = **direct hardware access**

Learn:

1. Pointer basics

```c
int x = 10;
int *ptr = &x;
```

2. Dereferencing

```c
*ptr = 20;  // changes x
```

3. Pointer arithmetic
4. Function pointers (used for ISR handlers)
5. Memory-mapped I/O

```c
#define UART0_DR *((volatile uint32_t*)0x4000C000)
UART0_DR = 'A'; // send character
```

Exercises:

- Create a mock memory map in an array and access it with pointers

- Write a function to toggle bits in a register using pointer arithmetic

---

Day 7: Structures and Hardware Mapping

Hardware registers are often grouped in structures.

Learn:

- `typedef struct` for grouping registers
- `#define` to map to hardware address
- Why volatile structs matter

Example:

```
typedef struct {
    volatile uint32_t DATA;
    volatile uint32_t DIR;
    volatile uint32_t IS;
} GPIO_TypeDef;

#define GPIOA ((GPIO_TypeDef *)0x40020000)

GPIOA->DIR = 0xFF; // Set pins as output
GPIOA->DATA = 0xAA; // Toggle pins
```

Exercise:

- Map at least one peripheral using a struct
- Write functions to set/clear/toggle bits in that struct

Week 2 — Control Flow, Functions, and Bit Manipulation

Day 1–2: Control Flow for Embedded Logic

Embedded firmware uses deterministic logic:

- `if`, `else`
- `switch` statements for state machines
- `for`, `while` loops (often with precise timing)
- Avoid recursion (stack constraints)

Exercise:

- Implement a small **state machine** for LED blink patterns
- Each state represents a hardware condition

Day 3–4: Functions and Modularization

- Write modular code for readability and reuse
- Pass pointers to functions for hardware manipulation
- Understand `inline` functions for performance

Example:

```
void set_led(GPIO_TypeDef *gpio, uint8_t mask) {
    gpio->DATA |= mask;
}
```

Exercise:

- Write `set_led`, `clear_led`, and `toggle_led` functions using struct pointers

Day 5–6: Bit Manipulation in Depth

- Masking:

  ```
  reg |= (1 << 3); // set bit 3
  reg &= ~(1 << 3); // clear bit 3
  reg ^= (1 << 3); // toggle bit 3
  ```

- Bitfields:

  ```
  typedef struct {
      uint32_t BIT0:1;
      uint32_t BIT1:1;
      uint32_t RESERVED:30;
  } REG_BITS;
  ```

- Embedded importance: **everything is a bit in hardware**

Exercise:

- Create a mock 32-bit register and write functions to set, clear, and toggle specific bits

---

Day 7: Volatile and const Qualifiers

- `volatile` for registers or shared memory
- `const` for read-only hardware values
- Combined: `volatile const` for read-only hardware registers

Exercise:

- Map a read-only status register and read bits safely
- Try to write to it and observe compiler error/warning

Week 3 — Embedded Practice Mini-Projects

This week you **tie C basics to hardware control**.

Project 1: GPIO LED Driver

- Use struct-mapped GPIO
- Implement `set`, `clear`, `toggle` functions
- Blink LEDs in a state machine pattern
- Use volatile variables

Project 2: Timer Driver Simulation

- Simulate a timer register in memory
- Increment counter in a loop
- Trigger "interrupt" via function pointer callback
- Understand race conditions and volatile usage

Project 3: UART Simulation

- Map TX/RX registers as structs
- Implement `send_char()`, `receive_char()`
- Use pointer arithmetic to access buffer

End of Phase 1 Deliverables (End of Week 3)

By the end of Phase 1:

1. You can **declare and use exact-width types**

2. You can **use pointers to manipulate memory-mapped registers**
3. You can **structure code using structs, functions, and bitfields**
4. You can **manipulate bits and write deterministic hardware logic**
5. You've completed mini projects simulating GPIO, timers, and UART

✅ **Outcome:** You can now write **real embedded C drivers**, without touching HAL libraries. You are ready for **Phase 2 — Hardware-Oriented C: Writing Real Drivers on the Board**.

If you want, I can **start Phase 2 next**, covering **driver development for GPIO, UART, SPI, timers, and interrupts on a real LRU board** with **multi-card communication**, week-by-week, day-by-day, including testing and debugging.

Do you want me to go ahead with Phase 2?