A Guide to C Pointers for Embedded Systems

Mastering pointers is arguably the most critical step in moving from a novice to a proficient C programmer, especially for embedded systems. Pointers give you the power to work directly with memory, which is essential for interacting with hardware, managing limited resources, and writing efficient code.

The Core Idea: A Pointer is an Address

Think of your computer's memory as a giant street with millions of houses. Each house has a unique, numbered address.

- A **variable** is like the **house itself**, which contains your data (e.g., the number 99).
- A **pointer** is a special piece of paper where you've written down the **address of a house**. The pointer doesn't hold the data itself, just the location of the data.

This is powerful because instead of passing around large, bulky data structures, you can just pass around their addresses, which are small and efficient.

The Two Essential Operators: & and *

1. **The "Address-Of" Operator (&)**: Use this to **get the address** of a variable. It's like looking up a house's address to write down on your paper.
2. **The "Dereference" Operator (*)**: Use this to **go to the address** written on your paper and access the data inside that house.

Let's see this in a simple program:

```
#include <stdio.h>

int main() {
   int house_value = 99; // The house with the value 99 inside.

   // Declare a pointer. 'int *' means "a pointer that holds the
   // address of an integer variable".
   int *address_paper;

   // Use '&' to get the address of 'house_value' and store it
   // in our pointer.
   address_paper = &house_value;
```

```
    // Let's examine what we have:
    printf("Value inside the house: %d\n", house_value);
    printf("Address of the house: %p\n", (void*)&house_value);
    printf("What's written on our address paper: %p\n", (void*)address_paper);

    // Now, let's use '*' to dereference the pointer. This means we
    // "follow the address" to see the value at that location.
    printf("Using the paper to find the value in the house: %d\n", *address_paper);

    // We can even use the pointer to CHANGE the value in the house!
    // Go to the address on the paper and change the value there to 100.
    *address_paper = 100;
    printf("The house value has been changed to: %d\n", house_value);

    return 0;
}
```

Use Case 1: Modifying Variables in Functions (Pass-by-Reference)

The Problem: You want a function to modify a variable that was declared in another function (e.g., in main). By default, C passes *copies* of variables to functions, so the original is unaffected.

The Solution: Pass the *address* of the variable (a pointer) to the function. The function can then use that address to find and modify the original variable.

Example: Swapping two numbers.

```
#include <stdio.h>

// The function takes addresses (pointers) as input.
void swap_values(int *address_of_a, int *address_of_b) {
    // 1. Go to the address of 'a' and get its value, store it in temp.
    int temp = *address_of_a;

    // 2. Go to the address of 'b', get its value, then go to the
    //    address of 'a' and put that value there.
    *address_of_a = *address_of_b;
```

```
    // 3. Go to the address of 'b' and put the stored temp value there.
    *address_of_b = temp;
}

int main() {
    int x = 10;
    int y = 20;
    printf("Before swap: x = %d, y = %d\n", x, y);

    // We pass the ADDRESSES of x and y to the function.
    swap_values(&x, &y);

    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

Embedded Use Case: A function like read_sensor(float *temperature) needs to update the temperature variable in your main loop. It must take a pointer to do so.

Use Case 2: Pointers and Arrays

In C, the name of an array is essentially a constant pointer to its first element. This makes pointers a natural and efficient way to work with arrays.

Pointer Arithmetic: When you add a number to a pointer, it moves forward by that many *elements*, not that many bytes. The compiler automatically calculates the step size based on the pointer's type (e.g., sizeof(int)).

Example: Summing an array using only pointer notation.

```
#include <stdio.h>

long sum_array(int *arr_ptr, int size) {
    long total = 0;
    // Create a pointer that marks the end of our work area.
    int *end_of_array = arr_ptr + size;
```

```
    // Loop while our current position pointer is before the end marker.
    while (arr_ptr < end_of_array) {
        total += *arr_ptr; // Add the value at the current address.
        arr_ptr++;         // Move pointer to the next integer's address.
    }
    return total;
}

int main() {
    int my_numbers[5] = {10, 20, 30, 40, 50};
    long sum = sum_array(my_numbers, 5); // Pass the array (which is a pointer)
    printf("The sum is: %ld\n", sum);
    return 0;
}
```

Embedded Use Case: You receive a data packet from a radio into a character array (a buffer). You use a pointer to walk through the buffer, parsing the header, payload, and checksum byte by byte.

Intermediate Topic: Function Pointers

A function pointer stores the memory address of a function. This allows you to treat functions like variables—you can pass them to other functions, store them in arrays, and call them dynamically.

Example: A flexible calculation function.

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

// This function's third argument is a pointer to a function.
// The syntax 'int (*operation)(int, int)' means:
// "a pointer named 'operation' that points to a function
// which takes two ints and returns an int."
int do_operation(int x, int y, int (*operation)(int, int)) {
    // Call the function using the pointer
```

```
    return operation(x, y);
}

int main() {
    // Pass the 'add' function as an argument
    int result1 = do_operation(10, 5, add);
    printf("10 + 5 = %d\n", result1);

    // Pass the 'subtract' function as an argument
    int result2 = do_operation(10, 5, subtract);
    printf("10 - 5 = %d\n", result2);

    return 0;
}
```

The Killer Embedded Use Case: Callbacks

This is a cornerstone of advanced embedded programming. You write a generic driver, but the application-specific logic is "called back" via a function pointer.

Imagine a generic button driver. The driver handles debouncing, but what happens on a press?

```
// --- In your generic button_driver.h ---
// Declare a function pointer type for clarity
typedef void (*button_callback_t)(void);

// The init function takes a pointer to the callback function
void button_init(button_callback_t on_press_callback);

// --- In your main application.c ---
#include "button_driver.h"

void toggle_system_led() {
    printf("LED Toggled!\n");
    // Code to toggle an LED would go here
}
```

```
void reset_the_system() {
    printf("System Resetting!\n");
    // Code to trigger a software reset
}

int main() {
    // Scenario 1: Use the button to toggle an LED.
    // We give the button driver the address of OUR function.
    printf("Initializing button to toggle LED...\n");
    button_init(&toggle_system_led);

    // When the driver detects a press, it will call toggle_system_led().

    // Scenario 2: Use the same button in a different product to reset.
    // printf("Initializing button to reset system...\n");
    // button_init(&reset_the_system);
}
```

This makes your driver reusable and decoupled from the application logic. If you can explain this concept and its use cases in an interview, you will demonstrate a deep understanding of practical C programming.