# Module 4: Pointers and Memory - The Heart of C (Days 14-17)

## 1. Philosophy Focus: Knowledge Graph & Inductive Teaching

This is the most important module in the course. Everything in C—strings, arrays, dynamic memory, and high-performance code—is built on a deep understanding of pointers. We will connect this new concept directly to your existing knowledge of variables. The inductive exercises are designed to make the abstract concept of a "memory address" feel tangible and concrete.

### **DAY 14: What is a Pointer? Memory Addresses**

**Goal:** Understand that every variable lives at a specific address in memory, and a pointer is simply a variable that *stores* a memory address.

1. A Mental Model: Your Computer's Memory

Imagine your computer's RAM as a giant street of houses. Each house has a unique address (like 123 Main St). Each house can store something inside (like the number 10).

* A regular variable (int x = 10;) is like the *contents* of the house.
* A **pointer** is like a piece of paper where you have *written down the address* of that house.

**2. Operators: & (address-of) and \* (dereference)**

* & (The Ampersand): We've seen this with scanf. It's the "address-of" operator. If x is a variable, then &x gives you its memory address.
* \* (The Asterisk): This is the "dereference" or "indirection" operator. If p is a pointer that holds an address, then \*p lets you access the value *at that address*. It means "go to the address stored in this pointer."

3. Inductive Example:

Let's see this in code. Compile and run:

#include <stdio.h>  
  
int main(void) {  
 int x = 10; // A regular integer variable.  
   
 // 'p' is a POINTER to an integer.  
 // It's a variable designed to hold the ADDRESS of an int.  
 int \*p;   
  
 p = &x; // Store the memory address of 'x' in the pointer 'p'.  
  
 printf("Value of x: %d\n", x);  
 printf("Address of x: %p\n", &x); // %p is for printing addresses (pointers)  
 printf("Value of p (it holds the address of x): %p\n", p);  
   
 // Now, let's use the pointer to access x's value  
 printf("Value at the address p points to: %d\n", \*p); // Dereference p  
  
 // We can even change x through the pointer!  
 \*p = 25; // "Go to the address stored in p, and put 25 there."  
 printf("\nAfter changing via pointer, new value of x: %d\n", x);  
  
 return 0;  
}

**4. Discovering the Pattern**

* **Declaration:** To declare a pointer, you use an asterisk: type \*pointer\_name;. For example, int \*p\_int; declares a pointer that can hold the address of an int. double \*p\_double; can hold the address of a double.
* **Assignment:** You assign an address to a pointer: p = &x;.
* **Usage:** You use the \* again to get the value *at* the address: value = \*p;.

**5. Day 14 Practice**

1. Declare a double variable price and assign it a value.
2. Declare a pointer to a double called ptr\_price.
3. Store the address of price in ptr\_price.
4. Print the value of price in two ways: once using the price variable directly, and once by dereferencing ptr\_price.
5. Change the value of price using only the pointer ptr\_price. Print the new value of price to confirm it changed.

### **DAY 15: Pointers and Functions**

**Goal:** Learn how to use pointers to allow a function to modify variables from the code that called it.

1. The Problem: Functions Make Copies

When you pass a regular variable to a function, the function gets a copy of its value.

#include <stdio.h>  
  
void try\_to\_increment(int num) {  
 num = num + 1;  
 printf("Value inside function: %d\n", num);  
}  
  
int main(void) {  
 int x = 10;  
 try\_to\_increment(x);  
 printf("Value inside main: %d\n", x); // Will this be 10 or 11?  
 return 0;  
}

Run this. You'll see x is still 10 in main. The function only modified its local copy.

2. The Solution: Passing a Pointer (the Address)

Instead of passing the value, we can pass the address of the variable. This allows the function to reach back into main's scope and modify the original variable. This is called "pass-by-reference."

**Inductive Example:**

#include <stdio.h>  
  
// The parameter is now a POINTER to an integer.  
void actually\_increment(int \*ptr\_num) {  
 // We must dereference the pointer to change the value  
 // at the address it holds.  
 \*ptr\_num = \*ptr\_num + 1;   
}  
  
int main(void) {  
 int x = 10;  
   
 // We must pass the ADDRESS of x using the '&' operator.  
 actually\_increment(&x);  
   
 printf("Value inside main is now: %d\n", x); // It's 11!  
 return 0;  
}

This is the **real reason** scanf needs the &. scanf is a function that needs to modify your variables, so you must pass it their addresses.

**3. Day 15 Practice**

1. **Swap Function:** Write a function void swap(int \*a, int \*b) that takes two integer pointers and swaps the values they point to.
   * In main, declare int x = 5; and int y = 10;.
   * Call swap(&x, &y);.
   * Print x and y after the call to confirm they have been swapped.
   * Hint: You'll need a temporary local variable inside your swap function. int temp = \*a;

### **DAY 16: Pointers and Arrays**

**Goal:** Understand the fundamental relationship between arrays and pointers in C.

1. The Big Reveal

In C, the name of an array is essentially a pointer to its first element.

**Inductive Example:**

#include <stdio.h>  
  
int main(void) {  
 int numbers[4] = {10, 20, 30, 40};  
  
 printf("Address of the first element (&numbers[0]): %p\n", &numbers[0]);  
 printf("Value of the array name (numbers): %p\n", numbers);  
   
 printf("\nThe first element (numbers[0]): %d\n", numbers[0]);  
 printf("The first element (\*numbers): %d\n", \*numbers);  
  
 return 0;  
}

When you run this, you will see that &numbers[0] and numbers print the *exact same address*. And numbers[0] and \*numbers print the same value.

2. Pointer Arithmetic

Because an array name is a pointer, you can do math with it. If you add 1 to a pointer, it doesn't add 1 byte; it moves forward in memory by the size of the data type it points to.

#include <stdio.h>  
  
int main(void) {  
 int numbers[4] = {10, 20, 30, 40};  
 int \*p = numbers; // p now points to the first element (10)  
  
 printf("First element: %d\n", \*(p));  
 printf("Second element: %d\n", \*(p + 1)); // Go to next int's address  
 printf("Third element: %d\n", \*(p + 2));  
   
 // In fact, array access `numbers[i]` is just syntactic sugar for `\*(numbers + i)`.  
  
 return 0;  
}

**3. Day 16 Practice**

1. **Sum an Array:** Write a function int sum\_array(int \*arr, int size).
   * The function should take a pointer to the start of an integer array and the number of elements in the array.
   * Inside the function, use a for loop to iterate through the array *using pointer arithmetic*.
   * In each iteration, add the value \*(arr + i) to a running total.
   * Return the total sum.
   * In main, create an array, call this function, and print the result.

### **DAY 17: Spaced Repetition - Review**

**Goal:** Solidify your understanding of pointers by using them in a practical function.

1. Done-for-you Training Plan: Find Min and Max

Write a function that finds both the minimum and maximum values in an array in a single pass. This is a common problem that perfectly demonstrates the power of passing pointers as function arguments.

**Project Requirements:**

1. Define a function with the following signature:  
   void find\_min\_max(int \*arr, int size, int \*out\_min, int \*out\_max)
2. **Parameters:**
   * int \*arr: A pointer to the beginning of an integer array.
   * int size: The number of elements in the array.
   * int \*out\_min: A pointer to an integer variable in the calling function. Your function will write the minimum value to this address.
   * int \*out\_max: A pointer to an integer variable where you'll write the maximum value.
3. **Implementation:**
   * Inside the function, assume the first element is both the min and the max. \*out\_min = arr[0]; and \*out\_max = arr[0];.
   * Loop through the rest of the array (from the second element to the end).
   * In each iteration, if you find a smaller number, update the value at out\_min (\*out\_min = ...).
   * If you find a larger number, update the value at out\_max.
4. **In main:**
   * Create an integer array, e.g., {5, 2, 9, 1, 8}.
   * Declare two regular integer variables, min\_val and max\_val.
   * Call your function, passing the array, its size, and the **addresses** of your min/max variables: find\_min\_max(my\_array, 5, &min\_val, &max\_val);.
   * After the function returns, print the values of min\_val and max\_val. They should be 1 and 9.