Gorgo C for C++ Programmers

April 11, 2026

6. Pointers

If you have been writing modern C++, you may have rarely (or never) used raw pointers. Smart pointers like `std::unique_ptr` and `std::shared_ptr` manage memory for you. References let you pass objects without copying them. The standard library hides pointer details behind iterators and containers.

In C, none of that exists. Pointers are everywhere, and you must be comfortable with them. Every dynamic data structure, every function that needs to modify its arguments, every interaction with the operating system — all involve pointers.

What Is a Pointer?

A pointer is a variable that holds a memory address. That's it. Instead of holding a value like `42`, a pointer holds the *location* where `42` is stored.

Declaring Pointers

A pointer type is declared by placing a `*` after the base type. The type before the `*` tells you what kind of data lives at the address the pointer holds:

```
int *p;         // p is a pointer to an int
char *s;        // s is a pointer to a char
double *d;      // d is a pointer to a double
```

Trap: The * belongs to the variable, not the type. This declaration creates one pointer and one regular int:

```
int *p, q;    // p is a pointer to int; q is just an int
```

To declare two pointers, you need two stars: `int *p, *q;`

The Address-Of Operator: &

The & operator returns the address of a variable. You have seen & in C++ for references — in C, it is strictly the address-of operator:

```
int score = 100;
int *p = &score;   // p now holds the address of score

printf("score = %d\n", score);    // 100
// something like 0x7ffd5e8a3b2c
printf("address of score = %p\n", (void *)p);
```

Dereferencing: *

The * operator on a pointer gives you the value at the address the pointer holds. This is called **dereferencing**:

```
int score = 100;
int *p = &score;

printf("Value at p: %d\n", *p);   // 100

*p = 200;                          // modify score through the pointer
printf("score is now: %d\n", score);  // 200
```

Notice the dual use of *: in a declaration, it means "this is a pointer." In an expression, it means "follow the pointer to the value."

Pointers to Pointers

Since a pointer is just a variable, it has an address too. You can create a pointer to a pointer:

```
int val = 42;
int *p = &val;       // p points to val
int **pp = &p;       // pp points to p

printf("val = %d\n", val);       // 42
printf("*p = %d\n", *p);        // 42
printf("**pp = %d\n", **pp);    // 42
```

You dereference `pp` twice: once to get `p`, and again to get `val`. Pointers to pointers show up frequently in C — for example, `main` can be declared as `int main(int argc, char **argv)`, where `argv` is a pointer to an array of string pointers.

Visualizing Pointers in Memory

Consider this small program:

```
int x = 1985;
int y = 80;
int *p = &x;
int **pp = &p;
```

Every variable lives at some address in memory. Here is what the layout looks like (using made-up but realistic addresses):

```
Variable    Address      Value
--------    ----------   ----------
x           0x1000       1985
y           0x1004       80
p           0x1008       0x1000  -------> x
pp          0x1010       0x1008  -------> p -------> x
```

The variable `x` lives at address `0x1000` and holds the value `1985`. The pointer `p` lives at address `0x1008` and holds the value `0x1000` — the address of `x`. The pointer-to-pointer `pp` lives at `0x1010` and holds `0x1008` — the address of `p`. Following the chain: `*pp` gives you `p` (which is `0x1000`), and `**pp` gives you `x` (which is `1985`).

Notice that `p` and `pp` are just variables that hold numbers. Those numbers happen to be memory addresses. There is nothing magical about a pointer — it is just a variable whose value is an address.

Tip: You can take the address of any variable with `&`, including the address of a pointer variable. The expression `&p` gives you the address where `p` itself is stored, not the address `p` points to.

NULL Pointers

A pointer that does not point to anything should be set to `NULL`:

```c
int *p = NULL;   // p points to nothing

if (p != NULL) {
    printf("Value: %d\n", *p);
} else {
    printf("Pointer is NULL\n");
}
```

Dereferencing a `NULL` pointer is undefined behavior and usually crashes your program with a segmentation fault. Always check before dereferencing a pointer you did not initialize yourself.

Tip: In C, `NULL` is typically defined as `((void *)0)`. You may also see `0` used directly. C++11 introduced `nullptr` as a type-safe null pointer — C does not have `nullptr`, so use `NULL`.

Pointers and Arrays

A pointer might point to a single value in memory, or it might point to the first element of an array of values. There is nothing in the type system that tells you which — an `int *` looks the same either way:

```c
int score = 100;
int *p = &score;              // points to one int

int nums[] = {10, 20, 30};
int *q = nums;                // points to the first of three ints
```

Both `p` and `q` are `int *`. The compiler does not know whether there are more `int` values after the one being pointed to. It is up to you, the programmer, to keep track of how many elements a pointer refers to and to stay within bounds.

You already saw in the Variables chapter that an array name decays to a pointer to its first element. Now let's see what that lets you do.

Pointer arithmetic works in units of the pointed-to type. If `p` is an `int *` and `int` is 4 bytes, then `p + 1` advances the address by 4 bytes to the next `int`. You never have to think about byte sizes — the compiler handles it:

```
int nums[] = {10, 20, 30, 40, 50};
int *p = nums;

printf("%d\n", *p);       // 10 (same as nums[0])
printf("%d\n", *(p + 1)); // 20 (same as nums[1])
printf("%d\n", p[2]);     // 30 — yes, you can use [] on pointers!
```

Tip: Array indexing `nums[i]` is actually syntactic sugar for `*(nums + i)`. This is why `2[nums]` technically works — it is `*(2 + nums)`, which is the same thing. Don't write code like that, but knowing this helps you understand how arrays and pointers relate.

Pointers and Structures

You already know how to declare structs and access members with `.` from the Variables chapter. Pointers to structures are extremely common in C — almost any non-trivial program passes struct pointers around rather than copying entire structs.

```
struct song {
    char title[40];
    int year;
};

struct song track = {"Karma Chameleon", 1983};
struct song *p = &track;
```

To access a field through a pointer, you must dereference the pointer first. But the `.` operator has higher precedence than `*`, so you need parentheses:

```
printf("Title: %s\n", (*p).title);   // parentheses required
printf("Year: %d\n", (*p).year);
```

Writing `(*p).field` everywhere is tedious. C provides the `->` operator as a convenient shorthand:

```
printf("Title: %s\n", p->title);     // same as (*p).title
printf("Year: %d\n", p->year);       // same as (*p).year
```

Tip: The `->` operator is simply `(*p).field` written more clearly. You will see `->` far more often than `(*p).` in real C code. If you have a pointer to a struct, reach for `->`.

Pass by Value (and Pointers as a Workaround)

In C++, you can pass arguments by reference using &:

```
void increment(int &x) { x++; }   // C++ — modifies the original
```

C does not have references. **All function parameters in C are pass by value** — the function receives a copy of the argument, not the original. If you want a function to modify a variable in the caller, you pass a *pointer* to it:

```
void increment(int *x) {
    (*x)++;   // dereference the pointer, then increment
}
```

```c
int main(void) {
    int score = 99;
    increment(&score);      // pass the ADDRESS of score
    printf("%d\n", score);  // 100
    return 0;
}
```

The function `increment` receives a *copy* of the pointer (the address), but since both the original and the copy point to the same memory, dereferencing either one reaches the same variable. This is how C simulates pass by reference.

Tip: Every time you see a function parameter with `*` in C, ask yourself: "Is this pointer here so the function can modify the caller's variable, or because it needs to access a block of memory (like an array)?" Often it is both.

Try It: Pointer Starter

This program demonstrates the core pointer operations:

```c
#include <stdio.h>

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

int main(void) {
    // Basic pointer usage
    int val = 1985;
    int *p = &val;
    printf("val = %d, *p = %d\n", val, *p);
    printf("Address of val: %p\n", (void *)&val);

    // Modify through pointer
    *p = 1989;
    printf("After *p = 1989: val = %d\n", val);

    // Pointer to pointer
    int **pp = &p;
    printf("**pp = %d\n", **pp);

    // Pass by value with pointers
    int x = 10, y = 20;
    printf("Before swap: x=%d, y=%d\n", x, y);
    swap(&x, &y);
    printf("After swap:  x=%d, y=%d\n", x, y);

    // Arrays and pointers
    const char *words[] = {"Totally", "Radical", "Tubular"};
    for (int i = 0; i < 3; i++) {
        printf("words[%d] = %s\n", i, words[i]);
    }
```

```
    return 0;
}
```

Key Points

- A pointer holds a memory address. Use `&` to get an address and `*` to dereference it.
- All pointers are the same size on a given system, regardless of the type they point to.
- Arrays decay to pointers in most expressions. `a[i]` is equivalent to `*(a + i)`.
- Pointer arithmetic moves in units of the pointed-to type, not bytes.
- Use `->` to access struct fields through a pointer. It is shorthand for `(*p).field`.
- All function parameters in C are pass by value. Pass a pointer to modify the caller's variable.

Exercises

1. **Think about it:** In C++, you can pass by reference to modify a caller's variable. Why do you think C was designed with only pass by value? What does this simplify in the language?
2. **What does this print?**

   ```c
   int a[] = {10, 20, 30, 40, 50};
   int *p = a + 2;
   printf("%d %d %d\n", *p, *(p - 1), p[1]);
   ```

3. **Calculation:** On a 64-bit system, what is `sizeof(int *)`, `sizeof(char *)`, and `sizeof(double *)`?
4. **Where is the bug?**

   ```c
   int *get_value(void) {
       int result = 42;
       return &result;
   }
   ```

5. **What does this print?**

   ```c
   int x = 10;
   int *p = &x;
   int **pp = &p;
   **pp = 20;
   printf("%d\n", x);
   ```

6. **Where is the bug?**

   ```c
   struct song {
       char title[40];
       int year;
   };

   struct song *p = NULL;
   printf("%s\n", p->title);
   ```

7. **Write a program** that declares an array of 5 integers, uses a pointer to iterate through the array, and prints each element along with its memory address.