

UNIT 9

Pointer

(6 Hours) — C Programming (CACS151)

9.1 Introduction

A pointer is a variable that stores the memory address of another variable, rather than storing a data value directly. Pointers are one of the most powerful and distinctive features of C, enabling direct memory manipulation, efficient array and string handling, dynamic memory allocation, and the ability for functions to modify variables that belong to their caller.

Every variable in a running program occupies some location in memory, identified by a unique address. A pointer simply captures and stores that address so the program can refer to the variable indirectly — hence the term “indirection.”

Why Pointers Matter

- ✓ Allow a function to modify variables that belong to the caller
- ✓ Make array and string processing fast and flexible
- ✓ Enable dynamic memory allocation for data whose size is not known in advance
- ✓ Form the basis of more advanced structures such as linked lists and trees

9.2 The Address (&) and Indirection (*) Operators

C provides two special unary operators to work with addresses and pointers:

- The address-of operator (&) returns the memory address of its operand.
- The indirection (dereference) operator (*) accesses the value stored at the address a pointer holds.

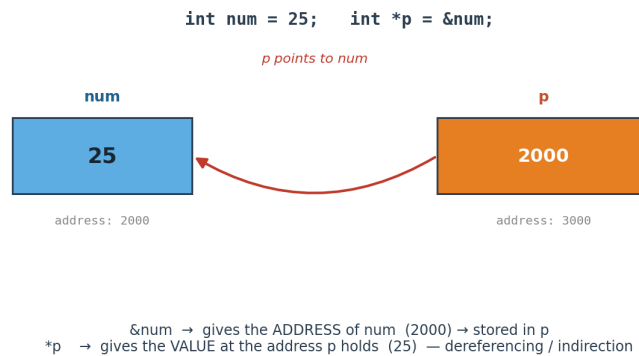


Figure 9.1 — Relationship between a variable, its address, and a pointer to it

Example — & and * Operators

```
int num = 25;
int *p = # /* p now holds the address of num */
printf("%d", num); /* prints 25 */
printf("%d", *p); /* also prints 25 — dereferencing p */
printf("%p", #); /* prints the address of num */
```

Note: $*p$ (read “value at address p”) and num refer to exactly the same memory location once $p = \#$ — changing one changes the other.

9.3 Declaration & Initialization

A pointer variable is declared by writing the asterisk (*) between the base data type and the pointer's name. The base type tells the compiler what kind of data the pointer will eventually point to, which is essential for correct pointer arithmetic and dereferencing.

Declaration	Meaning
<code>int *p;</code>	p is a pointer to an int (uninitialized — points nowhere valid yet)
<code>float *fp;</code>	fp is a pointer to a float
<code>char *cp;</code>	cp is a pointer to a char
<code>int *p = &x;</code>	p is declared and immediately initialized with the address of x
<code>int *p = NULL;</code>	p is initialized to point to nothing (a safe, defined “empty” state)

Note: A pointer should always be initialized — either with a valid address or with NULL — before it is dereferenced; dereferencing an uninitialized (“wild”) pointer leads to undefined behaviour.

9.4 Pointer to Pointer

Because a pointer is itself a variable stored somewhere in memory, it is possible to declare a pointer that stores the address of another pointer. This is called a pointer to pointer, declared with two asterisks (**).

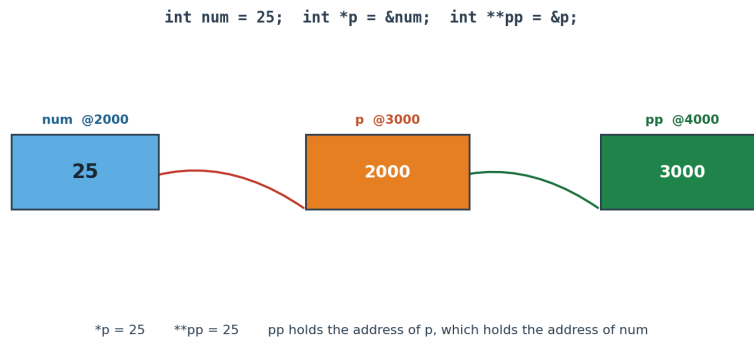


Figure 9.2 — A chain of pointers: pp points to p, and p points to num

Example — Pointer to Pointer

```
int num = 25;
int *p = #
int **pp = &p; /* pp holds the address of p */
printf("%d", **pp); /* dereference twice → 25 */
```

9.5 Pointer Expressions

Pointers can be combined with certain operators to form pointer expressions. Common valid operations include: assigning one pointer to another of the same type, comparing two pointers (`==`, `!=`, `<`, `>`), adding or subtracting an integer from a pointer, and subtracting one pointer from another of the same type (which yields the number of elements between them).

Example — Pointer Expressions

```
int a[5], *p1 = &a[0], *p2 = &a[4];
if (p1 < p2) printf("p1 comes before p2");
int diff = p2 - p1; /* diff = 4 (elements apart, not bytes) */
```

Note: Adding two pointers together, or multiplying/dividing a pointer, is not a meaningful or legal operation in C.

9.6 Pointer Arithmetic

When an integer is added to or subtracted from a pointer, C automatically scales the change by the size of the data type the pointer refers to, rather than simply moving by that many raw bytes. This makes pointer arithmetic naturally align with moving through an array element by element.

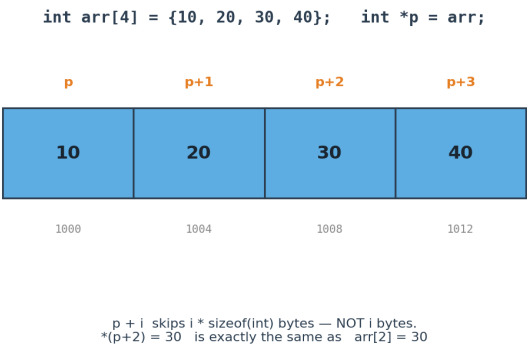
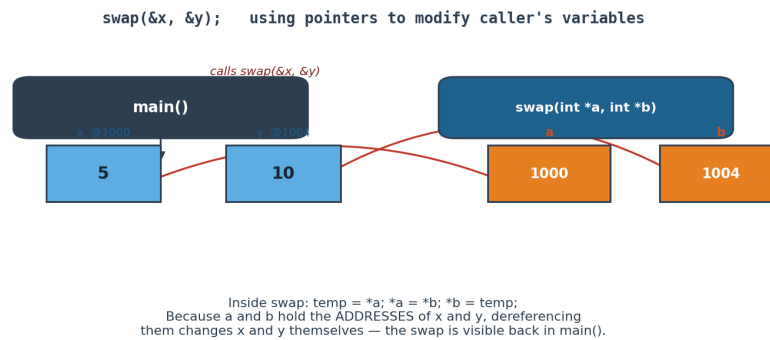


Figure 9.3 — Pointer arithmetic walking through an int array

Expression	Meaning
$p + 1$	Address of the next element (advances by $\text{sizeof}(*p)$ bytes)
$p - 1$	Address of the previous element
$*(p + i)$	Value of the element i positions after where p points — same as $p[i]$
$p++$	Moves p to point to the next element

9.7 Passing Pointer to a Function

By default, C passes arguments to functions by value — the function receives a copy, so changes made inside the function do not affect the caller's variables. Passing a pointer instead allows the function to receive the address of the caller's variable, so dereferencing that pointer inside the function directly modifies the original variable. This technique is often called call by reference (simulated in C using pointers).



Without pointers, a function only receives COPIES and cannot modify the caller's originals.

Figure 9.4 — Passing addresses to swap() so it can modify the caller's variables

Example — Swap Using Pointers

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

int x = 5, y = 10;
swap(&x, &y); /* x becomes 10, y becomes 5 */
```

9.8 Pointer and Array

Arrays and pointers are closely related in C. The name of an array, when used in most expressions, automatically decays into a pointer to its first element. Because of this, array elements can be accessed either with index notation or with equivalent pointer notation.

Array Notation	Equivalent Pointer Notation
arr[0]	*(arr) or *(p)
arr[i]	*(arr + i) or *(p + i)
&arr[i]	arr + i or p + i

Note: Although arr behaves like a pointer in expressions, arr itself is not a variable that can be reassigned — unlike a genuine pointer variable such as p, which can be made to point elsewhere.

9.9 Array of Pointer

Just as we can have an array of integers or an array of characters, we can also declare an array whose elements are themselves pointers. This is especially common for holding a list of strings, since each string is itself represented by a char pointer.

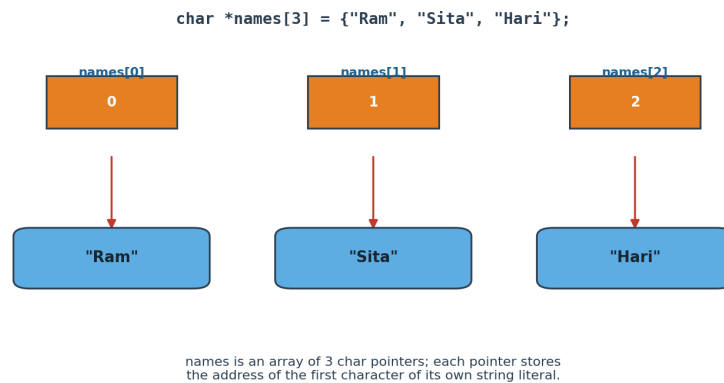


Figure 9.5 — An array of char pointers, each pointing to a separate string

Example — Array of Pointers to Strings

```
char *names[3] = {"Ram", "Sita", "Hari"};
for (int i = 0; i < 3; i++)
    printf("%s\n", names[i]);
```

9.10 Pointer and String

Since a C string is simply a character array terminated by '\0', a char pointer can be used to walk through and process a string one character at a time, exactly like pointer arithmetic on any other array.

Example — Traversing a String with a Pointer

```
char str[] = "Hello";
char *p = str;
while (*p != '\0') {
    printf("%c", *p);
    p++;
}
```

Note: A string literal such as "Hello" used directly in an expression is itself treated as a pointer to its first character.

9.11 Dynamic Memory Allocation

All the arrays discussed so far have a fixed size, decided at compile time. Dynamic memory allocation allows a program to request memory from the heap at run time, whenever the exact amount of memory needed is only known while the program is running. C provides four standard library functions, declared in `<stdlib.h>`, for this purpose.

Function	Purpose
<code>malloc(size)</code>	Allocates size bytes of uninitialized memory; returns a void* address
<code>calloc(n, size)</code>	Allocates memory for n elements of size bytes each, initialized to zero
<code>realloc(ptr, newSize)</code>	Resizes a previously allocated block to newSize bytes
<code>free(ptr)</code>	Releases previously allocated memory back to the system

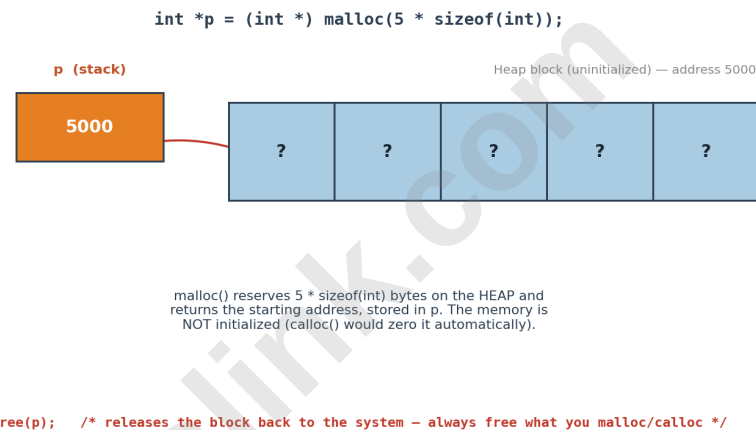


Figure 9.6 — `malloc()` reserving a block of heap memory referenced by a pointer

Example — Dynamic Array with malloc

```
int n = 5;
int *p = (int *) malloc(n * sizeof(int));
if (p == NULL) {
    printf("Memory allocation failed");
} else {
    for (int i = 0; i < n; i++) p[i] = i * i;
    free(p); /* release the memory when done */
}
```

Note: Every successful call to `malloc()` or `calloc()` should eventually be matched with a call to `free()` on the same pointer, to avoid memory leaks.

Key Takeaways — Unit 9

- ✓ A pointer stores the memory address of another variable; & gets an address, * dereferences it
- ✓ Pointer declarations must specify a base type, matching the kind of data pointed to
- ✓ A pointer to pointer (**) stores the address of another pointer, forming a chain of indirection
- ✓ Pointer arithmetic automatically scales by the size of the pointed-to type, not raw bytes
- ✓ Passing a pointer to a function lets the function modify the caller's original variable
- ✓ An array name decays into a pointer to its first element, so array and pointer notation are largely interchangeable
- ✓ An array of pointers is commonly used to store a list of strings
- ✓ Dynamic memory allocation (malloc/calloc/realloc/free) lets a program request and release heap memory at run time

— End of Unit 9 —

bcalink.com