

Arrays in C

A Study Handout

From Memory to Multidimensional Arrays

Sreetam Ganguly

August 2026

Copyright © 2026 Sreetam Ganguly.

This handout is free material: you may redistribute it and/or modify it under the terms of the **GNU General Public License version 3** as published by the Free Software Foundation, or (at your option) any later version.

It is distributed in the hope that it will be useful, but **without any warranty**; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details. You should have received a copy of the licence along with this document; if not, see <https://www.gnu.org/licenses/>.

Abstract

This handout builds an understanding of arrays in C from first principles. Rather than starting with array syntax, it begins with how a computer stores and addresses data in memory, then shows how variables, pointers, and arrays are all different ways of working with the same underlying memory. By the end, arrays should feel less like a new syntax to memorize and more like a direct consequence of how memory is organized.

Contents

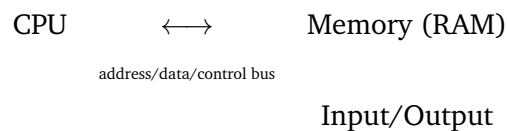
1	The Von Neumann Architecture	2
2	Memory as a Sequence of Addressed Bytes	2
3	Variables, Pointers, and Contiguous Memory	3
3.1	Variables are named memory	3
3.2	Pointers store addresses	3
3.3	Contiguous memory	3
4	Arrays: Definition and Memory Layout	4
4.1	Declaring and initializing arrays	4
4.2	Reading and writing elements	4
5	Multidimensional Arrays	5
6	Strings as Character Arrays	5
7	Arrays as Function Parameters	5
8	Common Mistakes	6
9	Practice Problems	6

1 The Von Neumann Architecture

Almost every general-purpose computer in use today, including the one running your C program, is built on the Von Neumann architecture, proposed by John von Neumann in 1945. Its defining idea is simple but far-reaching: both the program's instructions and the data it operates on are stored in the same memory, and are fetched by the processor in the same way.

A simplified view of this architecture has four components:

- **CPU (Central Processing Unit)** — fetches instructions, decodes them, and executes them. It contains the arithmetic logic unit (ALU) and a small set of registers.
- **Memory (RAM)** — stores both the program's instructions and the data (variables, arrays, etc.) the program works with.
- **Input/Output devices** — keyboard, disk, network, display, and so on.
- **Buses** — the wires connecting the CPU and memory, along which addresses, data, and control signals travel.



Key Point

Every variable, array, and instruction in your program lives at some address in RAM. When your program runs, the CPU repeatedly performs a **fetch–decode–execute** cycle: it fetches an instruction from memory, decodes what it means, executes it (possibly reading or writing more memory), and moves to the next instruction. Understanding that data is just bytes sitting at addresses in memory is the foundation for understanding pointers and arrays.

2 Memory as a Sequence of Addressed Bytes

RAM (Random Access Memory) can be thought of as one long row of storage cells, where each cell holds one byte (8 bits), and each cell has a unique numeric address. “Random access” means the CPU can read or write any address directly, in roughly constant time, rather than having to scan through memory sequentially.

```

1000    1001    1002    1003    1004    1005    1006
[ byte ][ byte ][ byte ][ byte ][ byte ][ byte ][ byte ]
  
```

A few consequences of this model matter directly for C:

1. Every byte has an address. Addresses are just numbers, usually shown in hexadecimal (e.g., `0x7ffe1234`).
2. A variable's type determines how many consecutive bytes it occupies. A `char` typically occupies 1 byte; an `int` typically occupies 4 bytes; a `double` typically occupies 8 bytes (exact sizes are platform-dependent).
3. The address of a variable is the address of its first byte. If an `int` named `x` starts at address 1000, it occupies bytes 1000–1003 on a system with 4-byte integers.

You can observe a variable's address directly in C using the address-of operator `&`:

```
1 #include <stdio.h>
2
3 int main(void) {
4     int x = 42;
5     printf("Value of x: %d\n", x);
6     printf("Address of x: %p\n", (void *)&x);
7     return 0;
8 }
```

Running this program will print some address such as `0x7ffd3a2c1c9c` — the exact value differs on every run, because the operating system decides where in RAM to place your program’s variables. What matters is that it has one, and C gives you direct access to it.

3 Variables, Pointers, and Contiguous Memory

3.1 Variables are named memory

When you write `int x = 42;`, you are asking the compiler to reserve 4 bytes of memory, label that location with the name `x`, and store the value 42 there. The name `x` is a convenience for the programmer; the compiler translates it into an address before the program runs.

3.2 Pointers store addresses

A pointer is a variable whose value is an address rather than an ordinary value such as a number or character. Pointers are declared with a `*`, and the type before the `*` tells the compiler what type of data lives at that address.

```
1 int x = 42;
2 int *p = &x; // p now holds the address of x
3
4 printf("%d\n", *p); // dereference: read the value at that address
```

`&x` means “the address of x”; `*p` means “the value stored at the address held by p.” These two operators are inverses of each other.

3.3 Contiguous memory

This is the idea that makes arrays possible. **Contiguous** means “occupying consecutive, back-to-back memory addresses, with no gaps.” When C allocates several variables of the same type as an array, it does not scatter them randomly across RAM — it places them one after another, in a single unbroken block.

This is fundamentally different from declaring several separate variables, which the compiler is free to place anywhere it likes:

```
1 int a = 1;
2 int b = 2;
3 int c = 3;
4 // a, b, c are NOT guaranteed to be adjacent in memory
5
6 int arr[3] = {1, 2, 3};
7 // arr[0], arr[1], arr[2] ARE guaranteed to be adjacent
```

4 Arrays: Definition and Memory Layout

An array in C is a fixed-size, contiguous block of memory that holds a sequence of elements, all of the same type. Declaring an array reserves one continuous block of memory large enough for all its elements.

```
1 int scores[5] = {70, 85, 90, 60, 100};
```

If this array starts at address 2000, and an `int` occupies 4 bytes on this system, the layout in memory looks like this:

Value	70	85	90	60	100
Address	2000	2004	2008	2012	2016
Index	scores[0]	scores[1]	scores[2]	scores[3]	scores[4]

Two facts follow directly from this layout:

- The array name refers to the address of its first element. `scores` and `&scores[0]` refer to the same address.
- **Indexing is just address arithmetic.** The address of `scores[i]` is computed as:

```
address of scores[i] = address of scores[0] + i × sizeof(element type)
```

This is exactly what the expression `scores[i]` compiles down to. In fact, `scores[i]` is defined in the C standard to mean `*(scores + i)` — “move `i` elements forward from the start, then read the value there.”

- This is also why array indices in C start at 0 rather than 1: index 0 means “zero elements forward from the start,” which is the first element itself.

4.1 Declaring and initializing arrays

```
1 int values[10]; // uninitialized: garbage values
2 int values[10] = {0}; // all elements set to 0
3 int values[5] = {1, 2, 3, 4, 5}; // explicit initializer list
4 int values[] = {1, 2, 3, 4, 5}; // size 5 inferred from list
```

Arrays declared without an initializer at file scope or as `static` are zero-initialized by the compiler; arrays declared as ordinary local variables inside a function are not automatically zeroed, and contain whatever bytes happened to already be in that memory.

4.2 Reading and writing elements

```
1 int values[5] = {10, 20, 30, 40, 50};
2
3 int first = values[0]; // read
4 values[2] = 99; // write: element at index 2 becomes 99
5
6 for (int i = 0; i < 5; i++) {
7     printf("values[%d] = %d\n", i, values[i]);
8 }
```

5 Multidimensional Arrays

A two-dimensional array in C, such as `int grid[3][4];`, is still stored as one contiguous block of memory — C simply lays each row out one after another, a scheme known as **row-major order**.

```
1 int grid[3][4] = {
2     {1, 2, 3, 4},
3     {5, 6, 7, 8},
4     {9, 10, 11, 12}
5 };
```

Memory layout (starting at address 0):

Elements:	1	2	3	4	5	6	7	8	9	10	11	12
Indices:	[0,0]	[0,1]	[0,2]	[0,3]	[1,0]	[1,1]	...	[2,3]				

The address of `grid[r][c]` is computed as:

```
address of grid[r][c] = address of grid[0][0] + (r × columns + c) × sizeof(element type)
```

In words: skip past `r` full rows, then move `c` elements into the current row. This is why the number of columns must be known when declaring or passing a 2D array — without it, the compiler cannot compute how far one row is from the next.

6 Strings as Character Arrays

C has no dedicated string type. A string is simply a `char` array whose contents end with a special null terminator, the character `'\0'` (byte value 0), which marks where the string ends.

```
1 char word[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
2 char word2[] = "Hello"; // compiler adds '\0' automatically
```

Memory:

H	e	l	l	o	\0
---	---	---	---	---	----

Functions such as `printf("%s", word)` and `strlen(word)` work by reading characters one by one, starting from the given address, and stopping as soon as they encounter the null terminator. This means `char word[6]` can only safely hold a 5-character word plus the terminator — forgetting to reserve space for `'\0'` is a very common bug.

7 Arrays as Function Parameters

When an array is passed to a function, C does not copy the entire array. Instead, the array **decays** into a pointer to its first element. This has two practical consequences.

```
1 void printArray(int arr[], int size) {
2     // arr is really an int *, pointing at the caller's data
3     for (int i = 0; i < size; i++) {
4         printf("%d ", arr[i]);
5     }
6 }
```

```

6 }
7
8 int main(void) {
9     int nums[5] = {1, 2, 3, 4, 5};
10    printArray(nums, 5); // no copy is made
11    return 0;
12 }

```

1. Changes made inside the function affect the original array, because the function is operating on the same memory, not a copy.
2. The size of the array is lost once it decays to a pointer — `sizeof(arr)` inside `printArray` gives the size of a pointer (commonly 8 bytes), not the size of the original array. This is why the size must always be passed as a separate argument.

8 Common Mistakes

- **Off-by-one errors.** An array declared as `int arr[5]` has valid indices 0 through 4. Writing `arr[5]` accesses memory one element past the end of the array.
- **Out-of-bounds access is not checked by C.** Unlike some other languages, C does not verify that an index is within range. Reading or writing `arr[5]` on a 5-element array does not raise an error — it reads or corrupts whatever memory happens to sit right after the array, which can cause unpredictable behavior far away from the actual mistake.
- **Uninitialized local arrays.** `int arr[10];` inside a function contains leftover bytes from whatever was previously stored at that memory, not zeros.
- **Forgetting space for the null terminator** in character arrays.
- **Assuming `sizeof` works after decay.** `sizeof(arr)` only gives the true array size in the scope where the array was declared, not inside a function it was passed into.

9 Practice Problems

1. If an `int` array `a` begins at address 5000 and `sizeof(int) = 4`, what is the address of `a[7]`?
2. Explain, in terms of memory, why `a[i]` and `*(a + i)` produce the same result in C.
3. Write a function `int sumArray(int arr[], int n)` that returns the sum of `n` integers in `arr`.
4. Declare a 4×4 integer array representing a tic-tac-toe board, and write a loop that prints it row by row.
5. Explain why the following function does not correctly report the size of the array passed to it, and describe how to fix the caller so the size is used correctly:

```

1 int countElements(int arr[]) {
2     return sizeof(arr) / sizeof(arr[0]);
3 }
4

```

6. Write a function that reverses an integer array in place (without allocating a second array).

7. A string buffer is declared as `char name[10];` and the program reads a 15-character name into it without any bounds checking. Explain what kind of problem this causes and why C does not prevent it.