LECTURE 4

Arrays and Sequential Storage

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

AGENDA

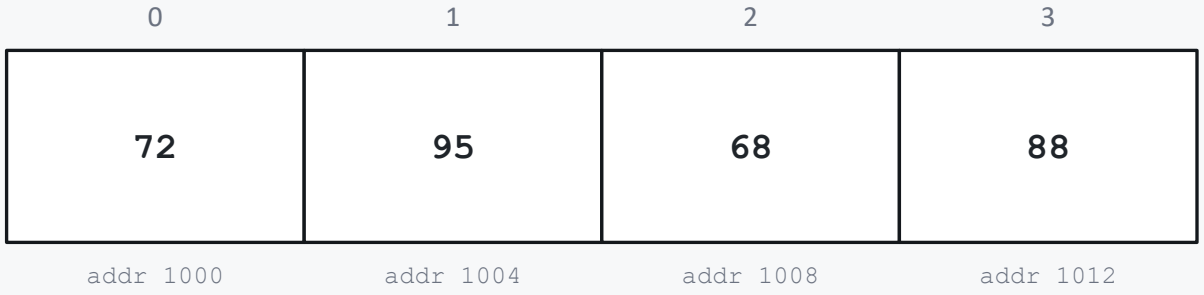
Learning Objectives

1. The array concept, and how it's actually laid out in memory
2. Indexing, and why array access is O(1)
3. One-dimensional and two-dimensional arrays, and row-major indexing
4. Why traversal order affects performance in practice — a real timed comparison
5. Traversal, insertion, deletion, searching, and updating on an array
6. The complexity of every array operation, and where arrays fall short

THE ARRAY CONCEPT

Contiguous Memory

An array stores elements of the same type back-to-back in memory, accessed by a numeric index. That contiguity is the entire reason arrays are fast.



scores[i] — each int is 4 bytes

If the computer knows the address of element 0 and the size of one element,

it can calculate the address of ANY element with simple arithmetic — no searching required.

THE ARRAY CONCEPT

The Address Formula — Why Access Is O(1)

```
address(scores[i]) =
    base_address + (i * size_of_element)

address(scores[2]) = 1000 + (2 * 4) = 1008
```

O(1)

one multiplication, one addition — regardless of array size

The raw address moves around from run to run (the OS places the stack differently each time), but the BYTE GAP between consecutive elements never changes — it's a structural guarantee of the array itself, verified directly against real compiled output: 4 bytes between scores[0] and scores[1], exactly sizeof(int).

ONE DIMENSION

One-Dimensional Arrays

one_d_array.cpp (excerpt)

```
int scores[5] = {72, 95, 68, 88, 91};

long byteGap = (char*)&scores[1] - (char*)&scores[0];
// byteGap == 4 == sizeof(int)   ->   yes, every run
```

The gap between elements is a property of the array itself, not of one run's coincidental addresses — that's why this course measures the gap, not the raw pointer value.

TWO DIMENSIONS

A 2D Array: Grade Book, 3 Students × 4 Quizzes

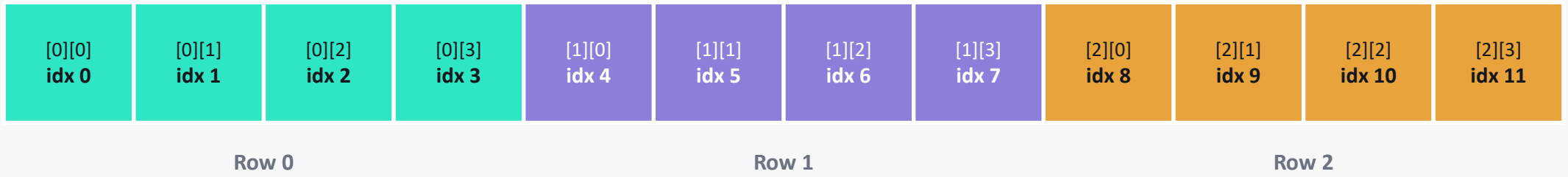
	Quiz 0	Quiz 1	Quiz 2	Quiz 3
Student 0	8	9	7	10
Student 1	6	8	9	9
Student 2	10	10	9	8

A 2D array is still ONE contiguous block in memory, row after row — quizScores[student][quiz] sits at base_address + (student * 4 + quiz) * size_of_int, just a slightly longer version of the same 1D formula.

TWO DIMENSIONS

Row-Major Order: Flattening the Grid

“Row-major” means the entire first row, then the entire second row, and so on — not column after column.




```
linear index = row × COLS + col
```

grid[1][2] → 1×4 + 2 = 6 — matches idx 6 above exactly.

The compiler will not stop you from indexing out of bounds — grid[5][0] compiles without warning and reads whatever memory happens to be there. That's undefined behavior, not a safe error.

Verifying the Formula Against Real Compiled Output

Element	Predicted offset	Actual offset	Match
grid[0][0]	0 bytes	0 bytes	yes
grid[0][3]	12 bytes	12 bytes	yes
grid[1][0]	16 bytes	16 bytes	yes
grid[1][3]	28 bytes	28 bytes	yes
grid[2][0]	32 bytes	32 bytes	yes
grid[2][3]	44 bytes	44 bytes	yes

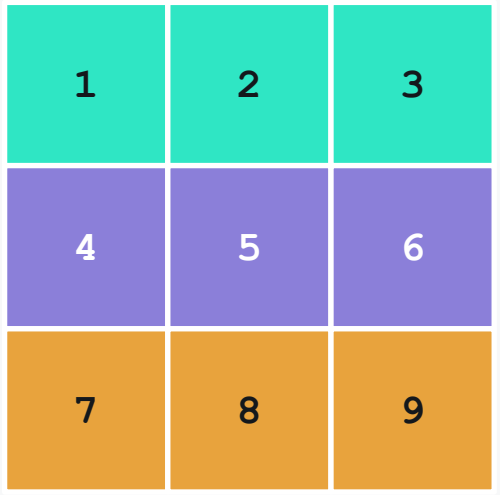
Every single prediction matches — the compiler lays out grid[3][4] exactly the way the hand-derived formula says it should.

CACHE PERFORMANCE

Why Traversal Order Matters

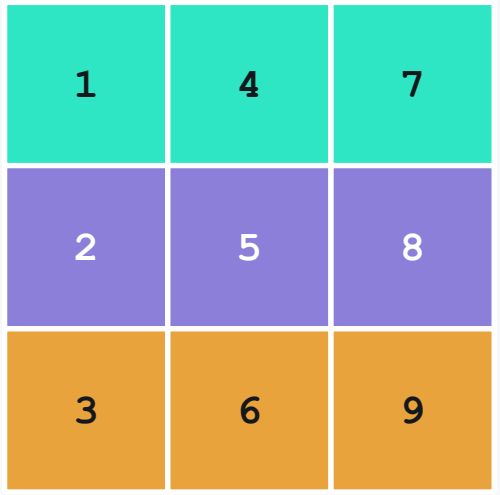
Same physical layout, same Big-O (visit every element once) — but a very different memory access pattern:

Row-major traversal — sequential



visits 1,2,3 within the same color (row) before moving on — cache-friendly

Column-major traversal — jumps every step



within one color (row) the numbers jump 1, 4, 7 — same cache line touched, abandoned, revisited later

The CPU fetches a whole cache line (~64 bytes) at once, betting nearby addresses will be needed next — row-major wins that bet on a row-major array.

PROOF, NOT JUST ASSERTION

Same O(n), Very Different Wall-Clock Time

Summing a 4000×4000 grid (16,000,000 elements) — identical work, same run, same machine:

38 ms

Row-major traversal

108 ms

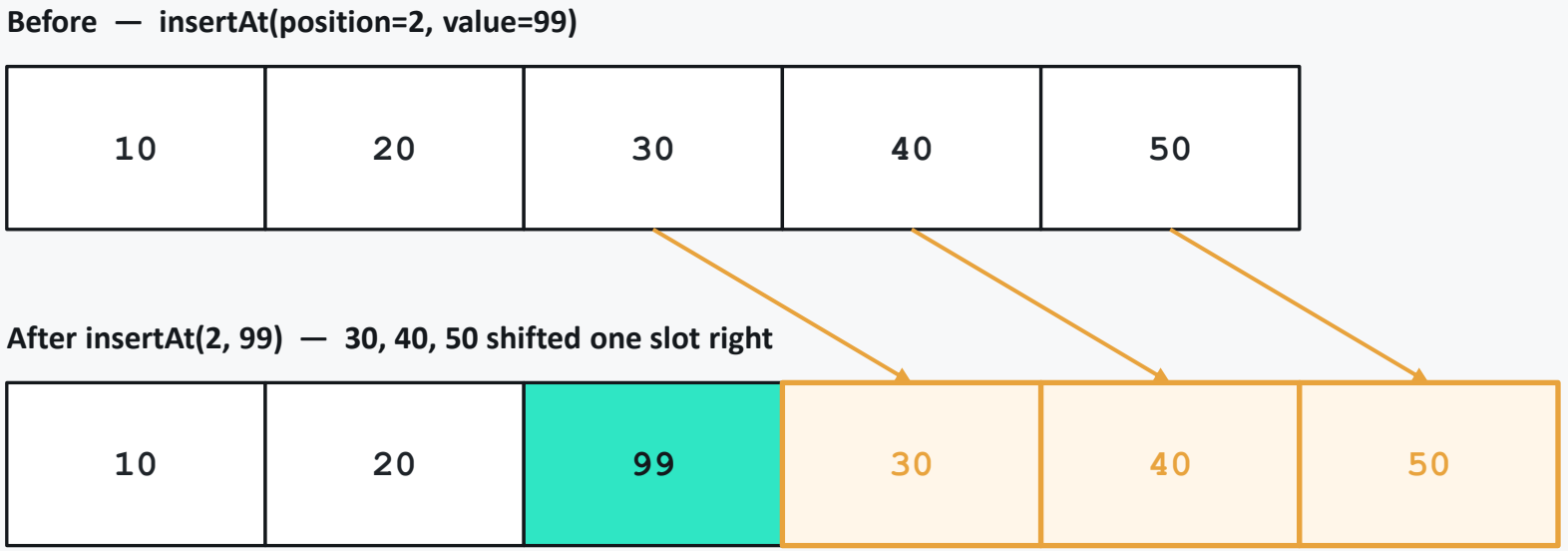
Column-major traversal — nearly 3× slower

Both functions perform exactly ROWS × COLS additions — identically O(n) in element count, and Big-O alone would call them the same. This is the gap between asymptotic complexity (Lecture 3) and real-world performance: Big-O tells you how an algorithm SCALES, but memory layout and cache behavior decide the constant factor hiding behind the O(...).

OPERATIONS

Insertion and Deletion: Where Arrays Start to Hurt

Elements must stay contiguous with no gaps — inserting in the middle means physically shifting every element after that position.



Inserting 99 at index 2 shifted 30, 40, 50 one place right — three moves for one insertion. In the worst case (inserting at the very front), EVERY existing element has to move.

OPERATIONS

insertAt and deleteAt

array_insert_delete.cpp (excerpt)

```
int insertAt(int arr[], int size, int position, int value) {
    for (int i = size; i > position; i--)
        arr[i] = arr[i - 1];        // shift right
    arr[position] = value;
    return size + 1;
}

int deleteAt(int arr[], int size, int position) {
    for (int i = position; i < size - 1; i++)
        arr[i] = arr[i + 1];        // shift left
    return size - 1;
}
```

OPERATIONS

Complexity of Array Operations

Operation	Complexity	Why
Access by index	O(1)	Direct address calculation, no searching
Update by index	O(1)	Same calculation, then a single write
Search (unsorted)	O(n)	Must check elements one by one, worst case
Insert at the end (spare capacity)	O(1)	No shifting needed
Insert at the start / middle	O(n)	Every later element must shift right
Delete from the end	O(1)	No shifting needed
Delete from the start / middle	O(n)	Every later element must shift left

TAKING STOCK

Advantages and Limitations of Arrays

Advantages

- O(1) random access to any element by index
- Memory-efficient — no extra per-element overhead (unlike a linked list's pointers)
- Cache-friendly — contiguous memory means the CPU loads several elements at once

Limitations

- Fixed size once created — growing past capacity means allocating a new array and copying everything
- Expensive insertion/deletion anywhere but the end
- Wastes memory if over-allocated, forces a costly resize if under-allocated

These exact limitations — fixed size, expensive middle insertion — are precisely what motivate the linked list, the subject of the next unit.

KEY TAKEAWAYS

Key Takeaways

- Arrays store elements in contiguous memory, which is exactly what makes indexed access O(1) — the address of any element is computed directly, never searched for.
- 2D arrays are stored the same contiguous way, row by row (row-major order), verified directly against the compiler's own computed offsets.
- Insertion and deletion in the middle cost O(n), because every following element must physically shift to stay contiguous — an array's central weakness.
- Two algorithms with identical Big-O can still run at very different real speeds — traversal order interacts with CPU cache behavior in a way Big-O alone doesn't capture.
- Arrays are fast and memory-efficient for read-heavy, rarely-resized data — and a poor fit when data needs to grow unpredictably or be inserted/removed from the middle often.