

# MODERN C++ DESIGN PATTERNS

Multi-Dimensional Arrays

by Prasanna Ghali

# Arrays: Memory Storage (1/2)

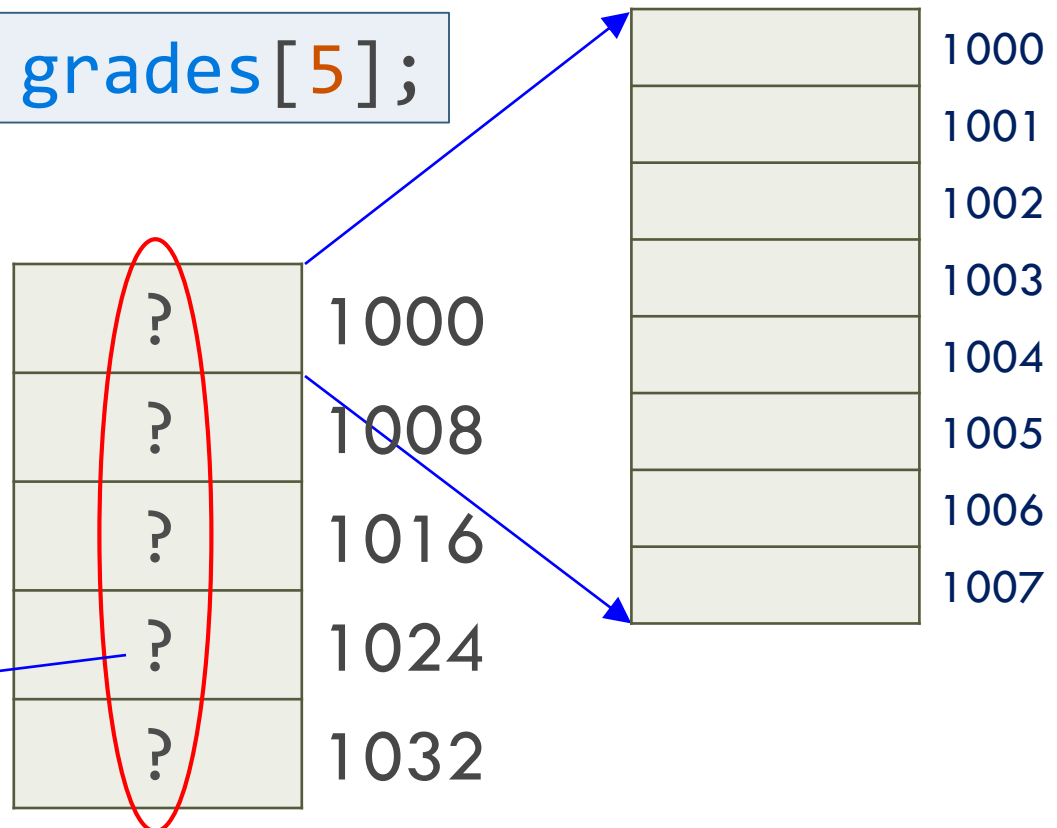
2

- Arrays have linear structure - elements are given contiguous memory storage

```
double grades[5];
```

Memory block is 40 bytes which is equivalent to  $\text{sizeof}(\text{double}) * 5$

No initializers during definition; thus, memory contents are garbage!!!

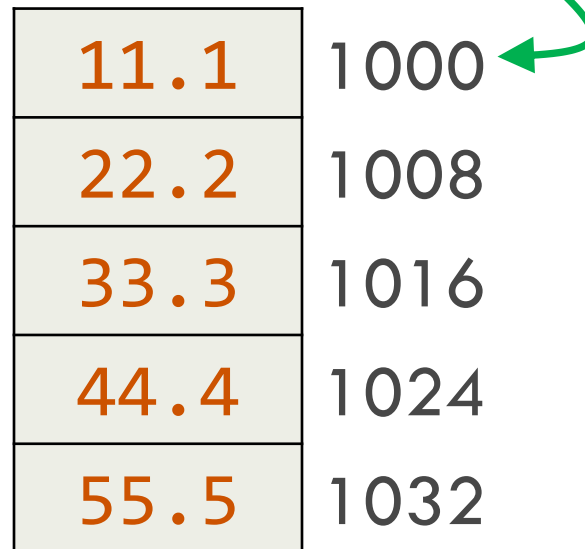


# Arrays: Memory Storage (2/2)

3

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

- 1) Base type **double** means each element is 8 bytes
- 2) Size 5 means contiguous memory block is 40 bytes
- 3) Compiler will fix *base address* for array, say 1000
- 4) From compiler's perspective, name **grades** means base address 1000



11.1	1000
22.2	1008
33.3	1016
44.4	1024
55.5	1032

# sizeof Operator

4

- For array names, `sizeof` returns number of bytes of storage for all array elements
- Type of value returned by `sizeof`: `size_t`
  - ▣ `size_t` is *largest unsigned integral type* – most implementations use `unsigned long int`

```
int    racers[5];  
double prizes[10];  
char   inits[1000];
```

Notice that array name is  
operand to `sizeof` operator



```
printf("sizeof(racers): %lu\n", sizeof(racers));  
printf("sizeof(prizes): %lu\n", sizeof(prizes));  
printf("sizeof(inits): %lu\n", sizeof(inits));
```

# Two-dimensional arrays:

## Introduction (1 / 3)

5

- One-dimensional arrays keep track of data values visualized as row or column
- Many examples (digital images, board games) exist where data is best visualized using grid or table having both rows and columns

A 3x4 grid representing a 2D array. The rows are labeled 'row 0', 'row 1', and 'row 2' on the left, with blue arrows pointing to each row. The columns are labeled 'column 0', 'column 1', 'column 2', and 'column 3' at the bottom, with blue arrows pointing to each column. The values in the grid are as follows:

row 0	2	0	7	1
row 1	3	-3	0	6
row 2	-1	5	3	4

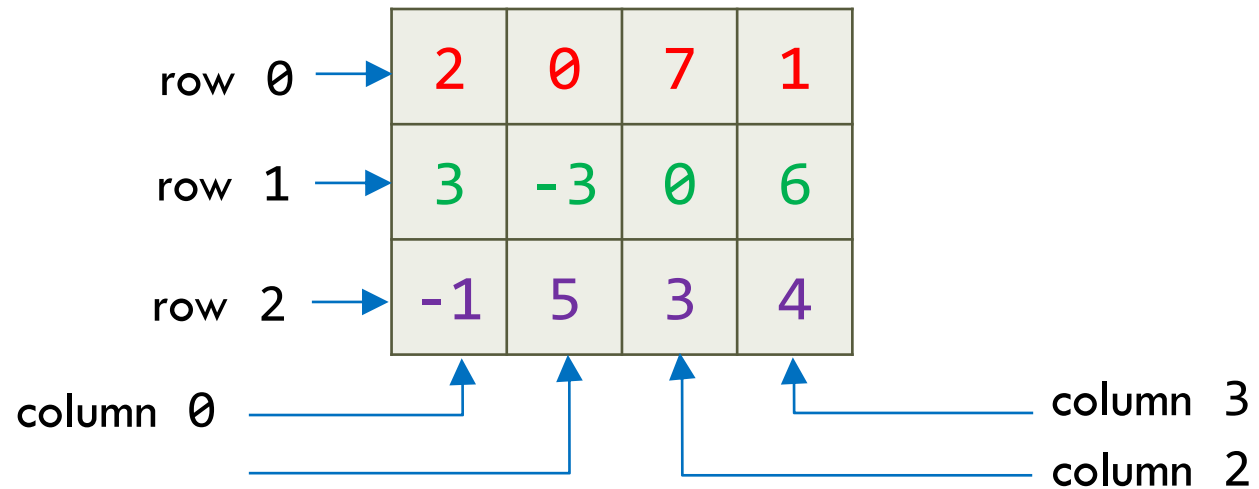
# Two-dimensional arrays:

## Introduction (2/3)

6

In C/C++, table or matrix represented as two-dimensional array: `int carrot[3][4];`

- `carrot` has 12 elements – each of type `int` – divided into 3 rows with each row having 4 columns



in memory

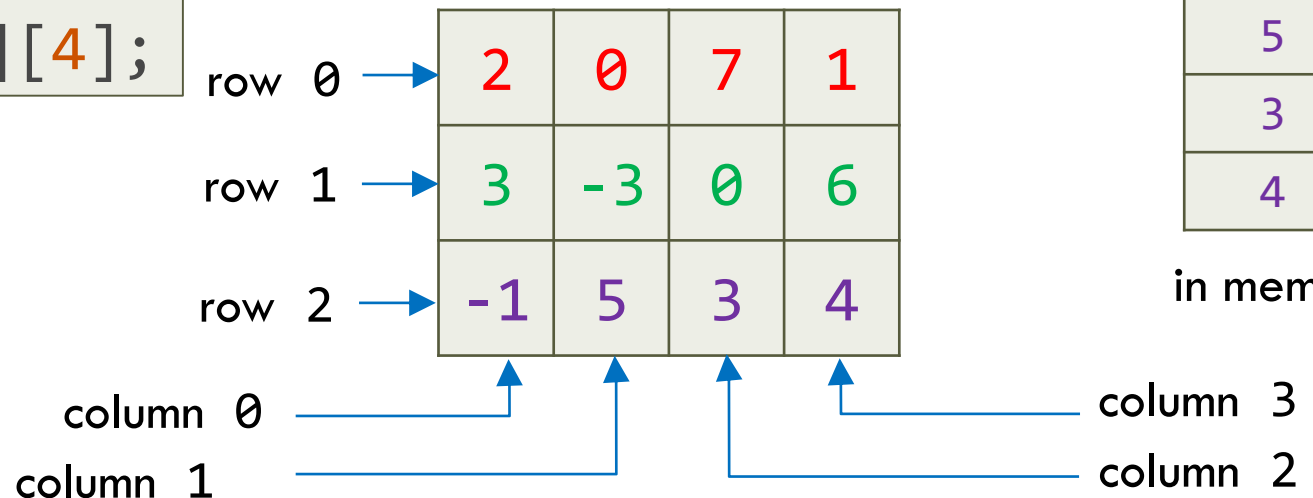
# Two-dimensional arrays:

## Introduction (3/3)

7

- Each element in `carrot` accessed using two subscripts – a *row* subscript and a *column* subscript
  - ▣ As usual, subscripts begin at zero
  - ▣ `carrot[1][2]` evaluates to `int` value 0

```
int carrot[3][4];
```



2
0
7
1
3
-3
0
6
-1
5
3
4

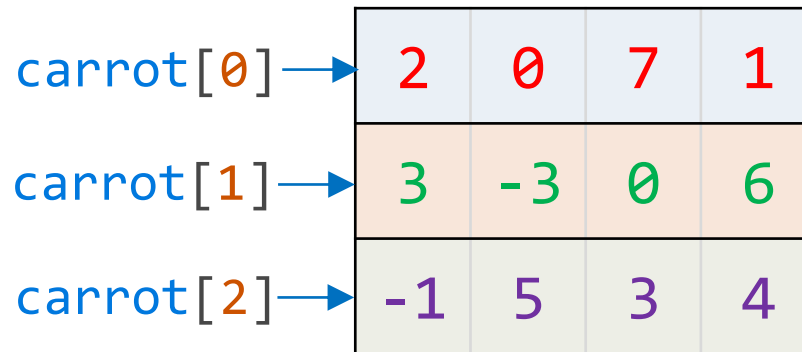
in memory

# Array of arrays (1 / 3)

8

- Internally, multi-dimensional arrays are considered as *array of arrays*
- Two-dimensional array is one-dimensional array with each element being one-dimensional array

```
int carrot[3][4];
```



subscript 0

subscript 1

subscript 3

subscript 2

2
0
7
1
3
-3
0
6
-1
5
3
4

in memory

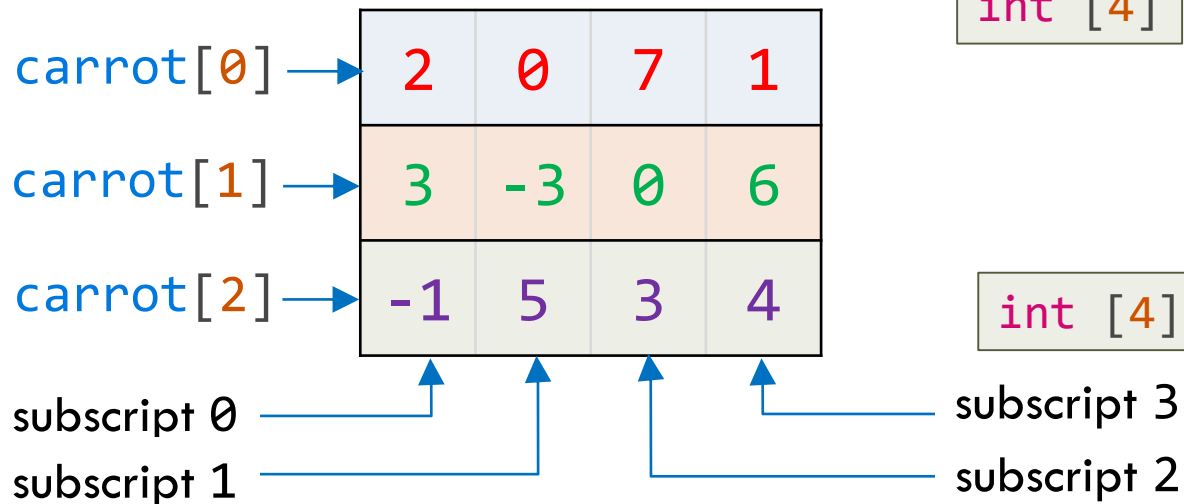


# Array of arrays (2/3)

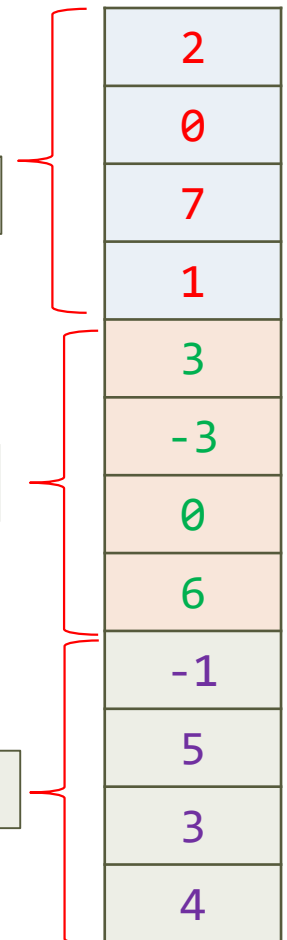
9

- Object `carrot` visualized as one-dimensional array of 3 elements, each element of type `int [4]`

```
int carrot[3][4];
```



in memory



# Array of arrays (3/3)

10

What does expression `carrot[2][1]` mean?

- Expression `carrot[2]` means 3<sup>rd</sup> element of array `carrot` having type `int [4]` (“array of 4 `ints`”)
- Expression `carrot[2][1]` means 2<sup>nd</sup> element in that array of 4 `ints`

```
int carrot[3][4];
```

2
0
7
1
3
-3
0
6
-1
5
3
4

in memory

<code>carrot[0]</code> →	2	0	7	1
<code>carrot[1]</code> →	3	-3	0	6
<code>carrot[2]</code> →	-1	5	3	4

$$9 \equiv 2 \times 4 + 1$$

subscript 0

subscript 1

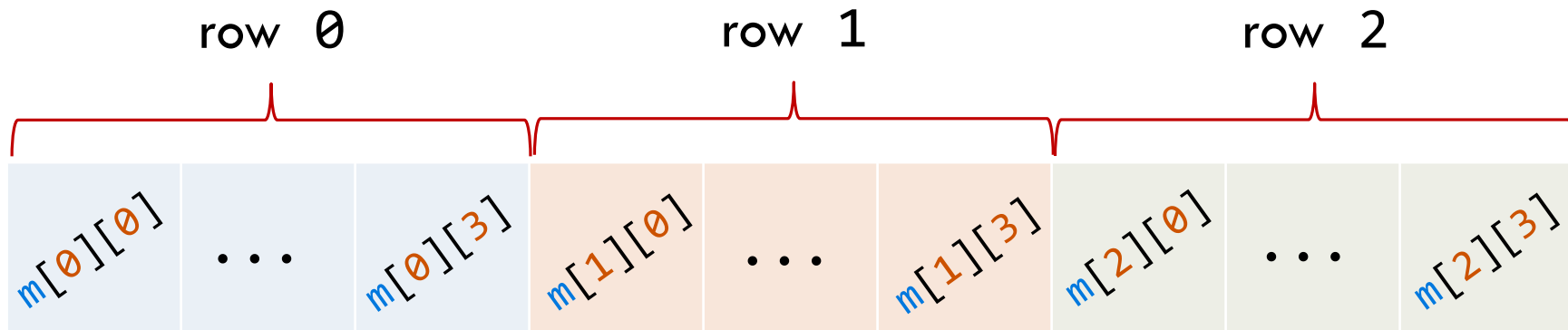
subscript 3

subscript 2

# Row-major storage

11

- Consider two-dimensional array `int m[3][4];`
  - ▣ 12 `int` elements of two-dimensional array `m` are contiguously stored in memory
  - ▣ Since `m` is array of arrays, 4 elements of 1<sup>st</sup> row `m[0]` are given contiguous storage, followed by 4 elements of 2<sup>nd</sup> row `m[1]`, and so on



# “Array of Array”-ish Declaration

## (1 / 2)

12

- C/C++ can be used to define a 3-by-4 multidimensional arrays of `ints` like this:

```
int carrot[3][4];
```

- Or in a way that is more “array of array”-ish like this:

```
using vegetable = int [4];  
vegetable carrot[3];
```

- In either case, individual element is accessed by `carrot[i][j]`
- At compile time, compiler will resolve that to

```
*(*(carrot+i)+j)
```

# “Array of Array”-ish Declaration

## (2/2)

13

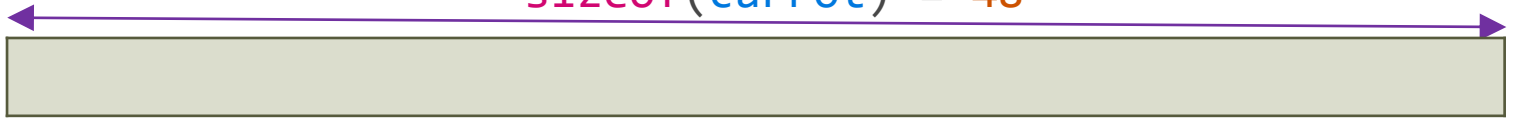
- Whenever you see “array” in C/C++, think “vector”, that is, a one-dimensional array of *something*, possibly another array

# Multidimensional Array Storage:

```
int carrot[3][4];
```

14

`sizeof(carrot) = 48`



```
int (*p)[4] = carrot
```

`sizeof(carrot[i]) = 16`



```
int *r = carrot[i]
```

`sizeof(carrot[i][j]) = 4`



```
int t = carrot[i][j]
```

# Multidimensional Array Storage:

```
int apricot[2][3][4];
```

15

$\text{sizeof}(\text{apricot}) = 96$

apricot

```
int (*p)[3][4] = apricot
```

$\text{sizeof}(\text{apricot}[i]) = 48$

apricot[0]

apricot[1]

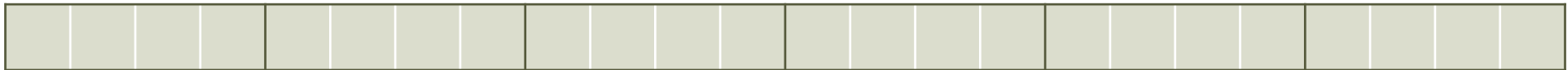
```
int (*r)[4] = apricot[i]
```

$\text{sizeof}(\text{apricot}[i][j]) = 16$

apricot[0][0] apricot[0][1] apricot[0][2] apricot[1][0] apricot[1][1] apricot[1][2]

```
int *t = apricot[i][j]
```

$\text{sizeof}(\text{apricot}[i][j][k]) = 4$



```
int u = apricot[i][j][k]
```