# HIGH-LEVEL PROGRAMMING I

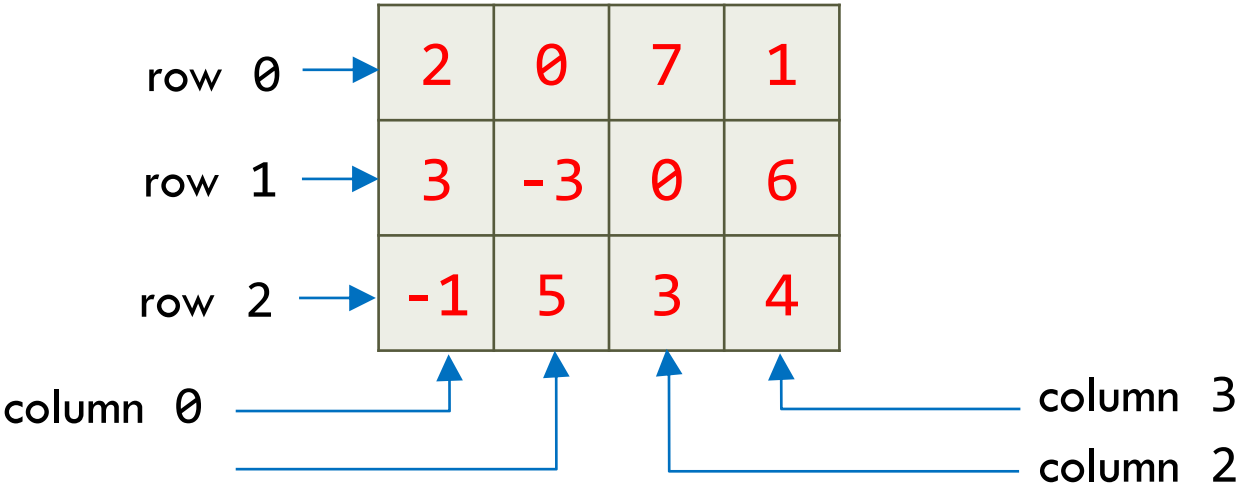
Multidimensional arrays

by Prasanna Ghali

# Two-Dimensional Arrays: Introduction (1/3)



- 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



# Two-Dimensional Arrays: Introduction (2/3)

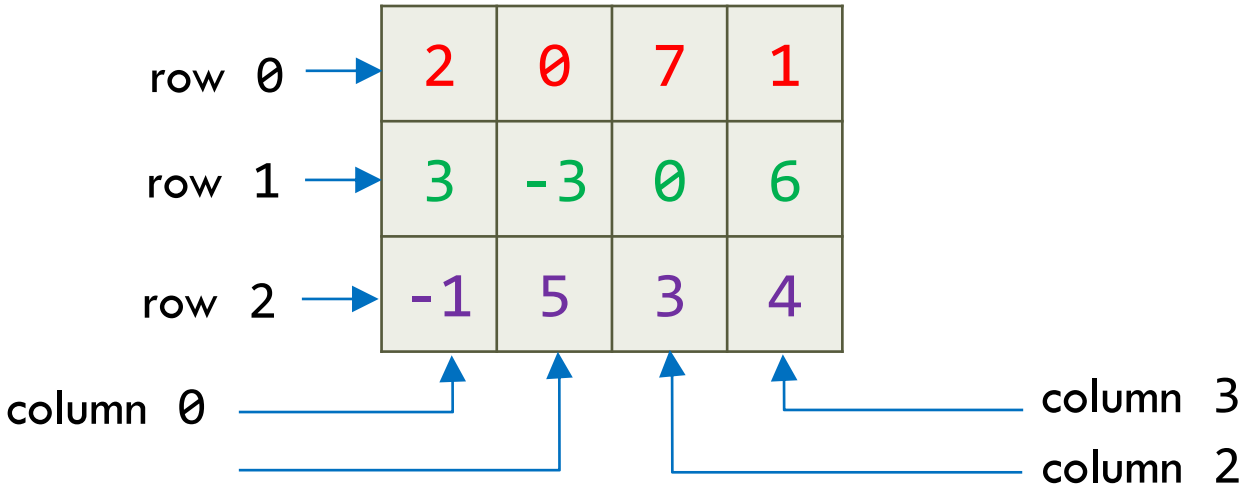


| 2 |
|---|
| 0 |
| 7 |
| 1 |
| 3 |
| -3 |
| 0 |
| 6 |
| -1 |
| 5 |
| 3 |
| 4 |

in memory

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

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

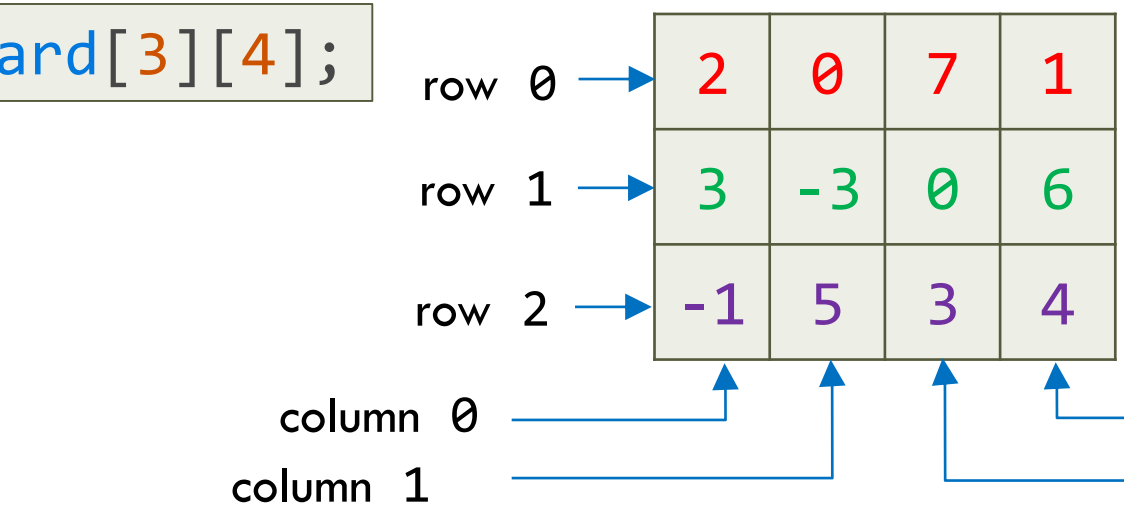


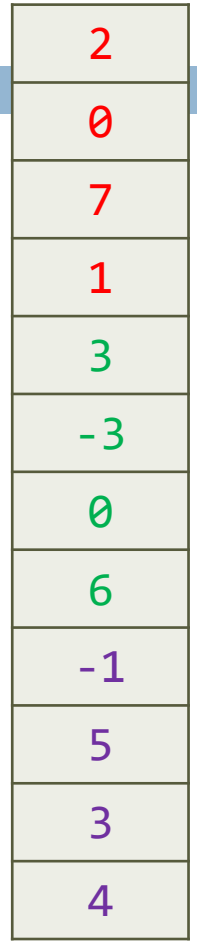
# Two-Dimensional Arrays: Introduction (3/3)



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

```
int board[3][4];
```



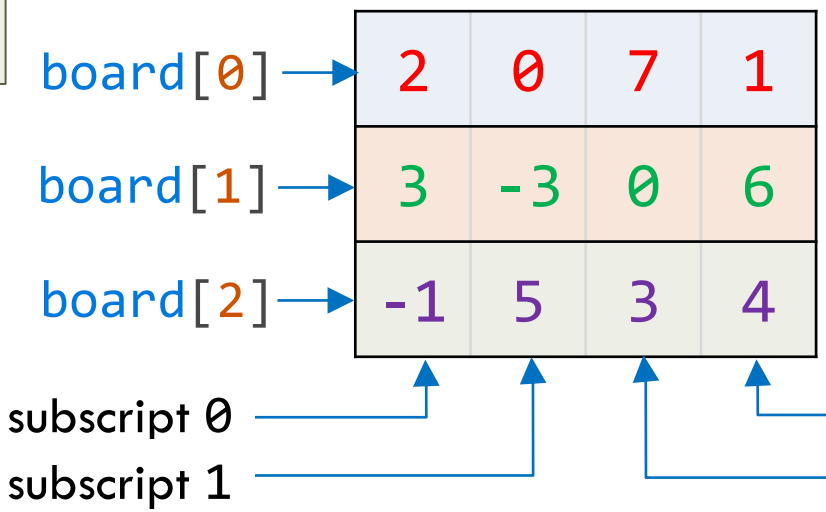

in memory

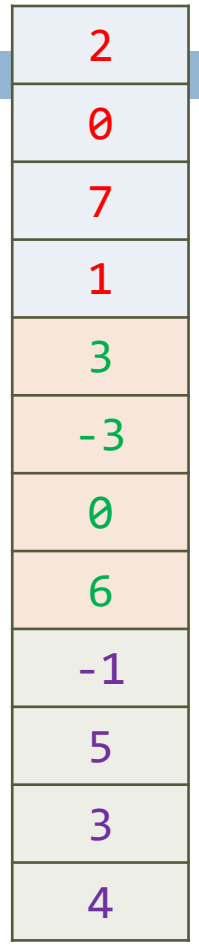
# Array of Arrays (1/3)



- 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 board[3][4];
```



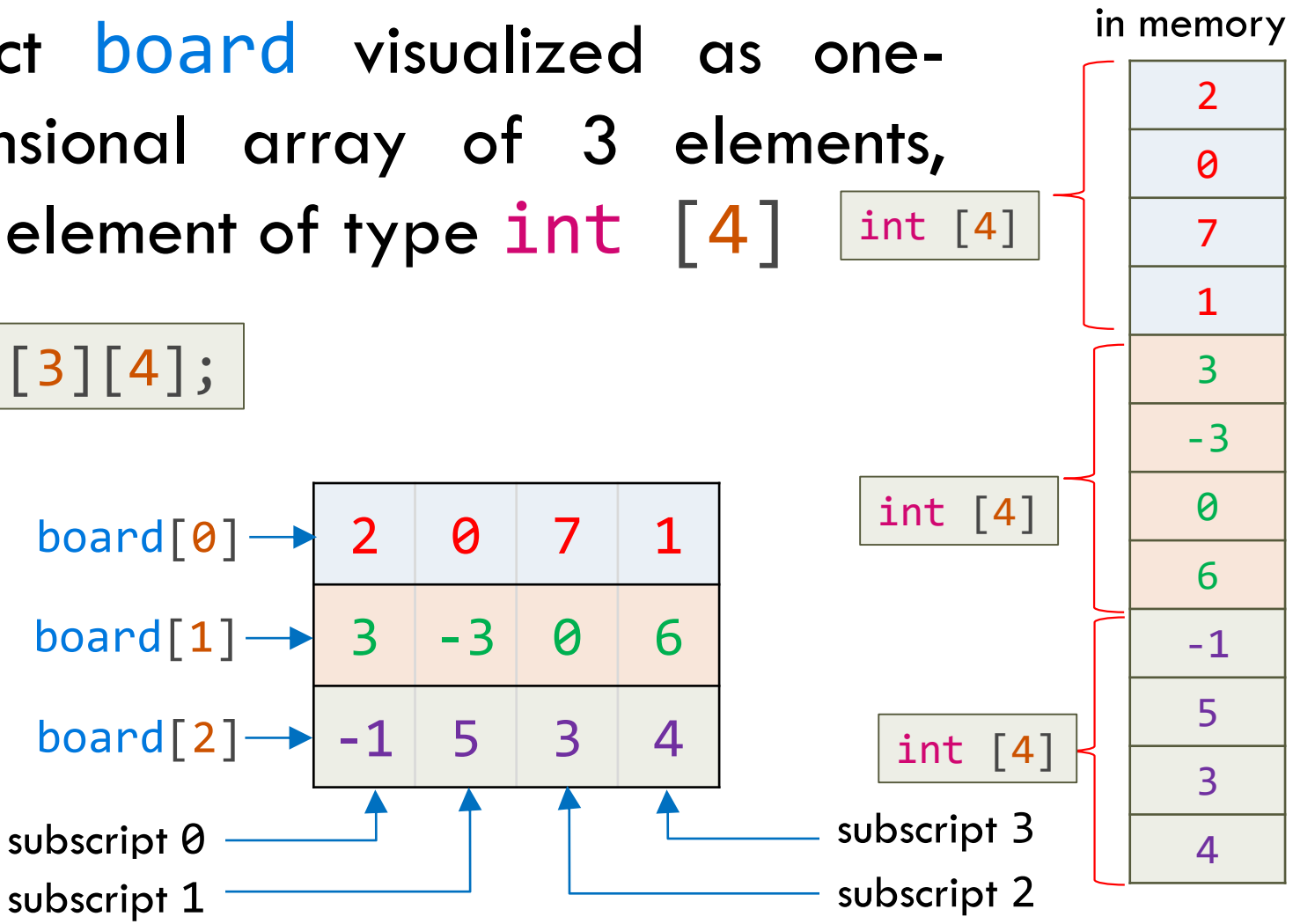

in memory

# Array of Arrays (2/3)



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

```
int board[3][4];
```



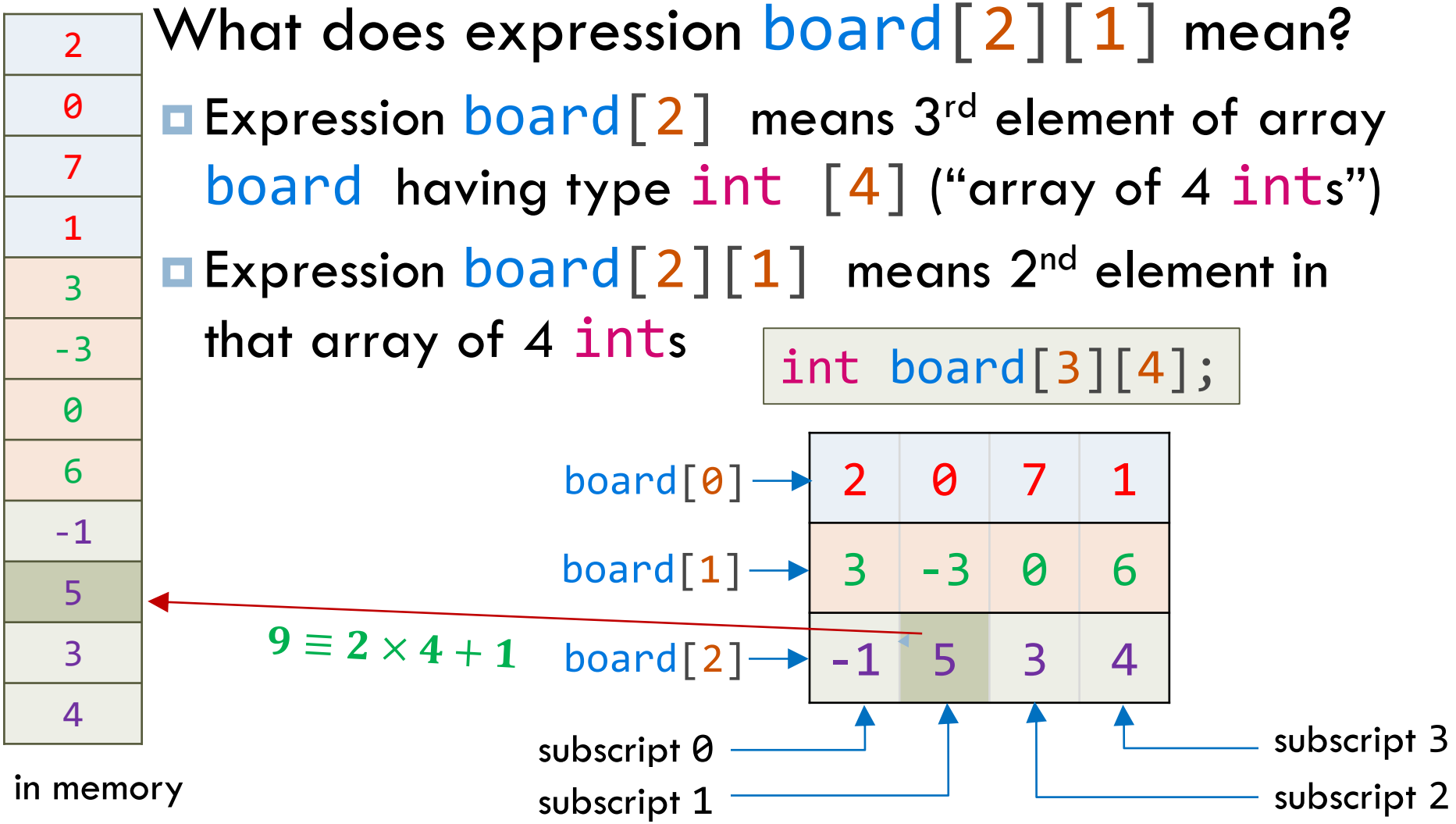
# Array of Arrays (3/3)



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

- Expression `board[2]` means 3rd element of array `board` having type `int [4]` ("array of 4 `int`s")
- Expression `board[2][1]` means 2nd element in that array of 4 `int`s

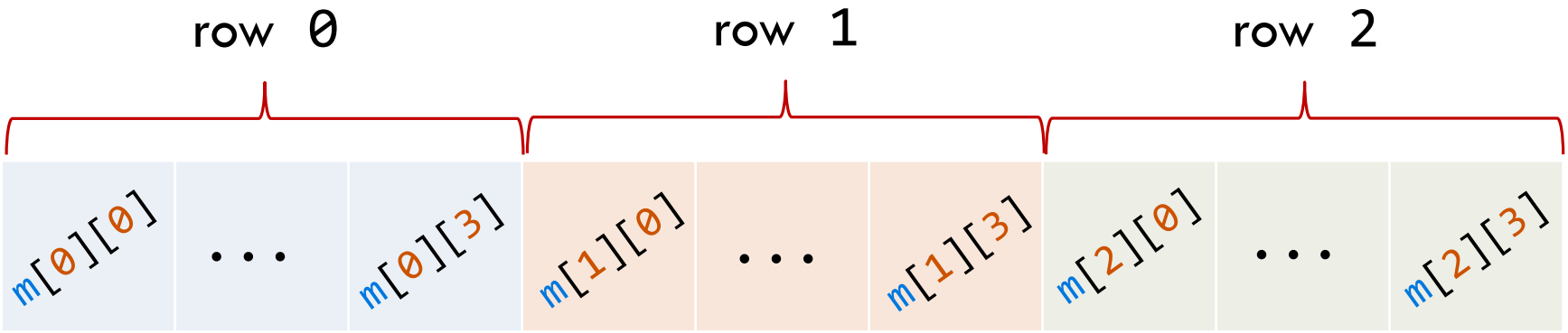
```
int board[3][4];
```



# Row-Major Storage



- 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 1st row `m[0]` are given contiguous storage, followed by 4 elements of 2nd row `m[1]`, and so on



# Initialization



- Equivalent ways of defining and initializing two-dimensional array

```
int board[3][4] = { {2, 0, 7, 1}, {3, -3, 0, 6}, {-1, 5, 3, 4} };
```

```
int board[3][4] = {
  { 2,  0, 7, 1},
  { 3, -3, 0, 6},
  {-1,  5, 3, 4}
};
```

```
int board[][4] = {
  { 2,  0, 7, 1},
  { 3, -3, 0, 6},
  {-1,  5, 3, 4}
};
```

# Abbreviated Initialization (1/4)



- If initializer count isn't enough to fill multidimensional array, remaining elements are zeroed

```
int board[3][4] = {
  2,
  3,
  -1,
};
```

```
int board[][4] = {
  {2},
  {3},
  {-1}
};
```

```
int m[][3] = { {1}, {2}, {3} };
```

```
int m[][3] = { 1, 2, 3 };
```

# Abbreviated Initialization (2/4)



- If initializer count isn't enough to fill multidimensional array, remaining elements are zeroed
  - Following initializer fills 1st three rows of `m`; remaining two rows are zeroed

```
int m[5][9] = {
  {1, 1, 1, 1, 1, 0, 1, 1, 1},
  {0, 1, 0, 1, 0, 1, 0, 1, 0},
  {0, 1, 0, 1, 1, 0, 0, 1, 0}
};
```

# Abbreviated Initialization (3/4)



- If inner list isn't long enough to fill a row, remaining elements in row are zeroed

```
int m[5][9] = {
  {1, 1, 1, 1, 1, 0, 1, 1, 1},
  {0, 1, 0, 1, 0, 1, 0, 1},
  {0, 1, 0, 1, 1, 0, 0, 1},
  {1, 1, 0, 1, 0, 0, 0, 1},
  {1, 1, 0, 1, 0, 0, 1, 1, 1}
};
```

# Abbreviated Initialization (4/4)



- Inner braces can be omitted
  - Once compiler sees enough values to fill one row, it begins filling the next

```
int m[5][9] = {
  1, 1, 1, 1, 1, 0, 1, 1, 1,
  0, 1, 0, 1, 0, 1, 0, 1,
  0, 1, 0, 1, 1, 0, 0, 1,
  1, 1, 0, 1, 0, 0, 0, 1,
  1, 1, 0, 1, 0, 0, 1, 1, 1
};
```

# Processing Multidimensional Arrays (1/3)



- Nested `for` loops ideal for processing multi-dimensional arrays

```
enum {NROWS = 4, NCOLS = 5};

int mat[NROWS][NCOLS];
for (int i = 0; i < NROWS; ++i) {
  for (int j = 0; j < NCOLS; ++j) {
    mat[i][j] = (i == j) ? 1 : 0;
  }
}
```

# Processing Multidimensional Arrays (2/3)



- Reading from a file that contains grades of 10 quizzes for 20 students

```
enum {NUM_STUDENTS = 20, NUM_QUIZZES = 10};

int grades[NUM_STUDENTS][NUM_QUIZZES];
FILE *stream = fopen("quizzes.txt", "r");
for (int i = 0; i < NUM_STUDENTS; ++i) {
  for (int j = 0; j < NUM_QUIZZES; ++j) {
    fscanf(stream, "%d", &grades[i][j]);
  }
}
```

# Processing Multidimensional Arrays (3/3)



- Finding day of year …

```
int day_of_year(int year, int month, int day) {
  static int daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
  };
  int leap = year%4==0 && year%100!=0 || year%400==0;

  for (int i = 1; i < month; ++i) {
    day += daytab[leap][i];
  }
  return day;
}
```

# Two-Dimensional Arrays and Function Parameters (1/2)



- If 2D array `grades` (from page 15) is to be passed to function, parameter declaration must include number of columns; number of rows is irrelevant

```
int max_quiz_grade(int a[][NUM_QUIZZES], int id) {
  int max_val = a[id][0];
  for (int j = 1; j < NUM_QUIZZES; ++j) {
    max_val = (a[id][j] > max_val) ? a[id][j] : max_val;
  }
  return max_val;
}
```

# Two-Dimensional Arrays and Function Parameters (2/2)



- Passing `grades` to a function

```
int grades[NUM_STUDENTS][NUM_QUIZZES];
FILE *stream = fopen("quizzes.txt", "r");
for (int i = 0; i < NUM_STUDENTS; ++i) {
  for (int j = 0; j < NUM_QUIZZES; ++j) {
    fscanf(stream, "%d", &grades[i][j]);
  }
}
// maximum quiz grade for student #5
int max_val = max_quiz_grade(grades, 4);
printf("max_val: %d\n", max_val);
```

# Pointer to Array (1/2)



- Given definition `int grades[20][10];`
- 2D array name `grades` is not pointer to `grades[0][0]` – it is pointer to `grades[0]`
  - `grades[i]` is pointer to `int[10]`
  - `(grades[i]+1)` will increment address given by `grades[i]` by number of bytes in `int[10]`

# Pointer to Array (2/2)



- Since 2D array `int grades[20][10];` is array of 20 elements – each element having type `int[10]` – parameter is "pointer to array of 10 `int`s"

```
int max_quiz_grade(int (*p)[NUM_QUIZZES], int id) {
  int max_val = p[id][0];
  for (int j = 1; j < NUM_QUIZZES; ++j) {
    max_val = (*(*(p+id)+j) > max_val) ? p[id][j] : max_val;
  }
  return max_val;
}

// function call with argument: int grades[10][20]
int max_val = max_quiz_grade(grades, 4);
```

# Two-dimensional Arrays as One-Dimensional Arrays (1/2)



- Can also view 2D array as one-dimensional array

```
int sum_grades(int const *begin, int const *end) {
  int sum = 0;
  while (begin < end) {
    sum += *begin++;
  }
  return sum;
}

// call to sum_grades for all students
int size = NUM_STUDENTS * NUM_QUIZZES;
int sum_val = sum_grades(&grades[0][0], &grades[0][0]+size);
printf("sum_val: %d\n", sum_val);
```

# Two-dimensional Arrays as One-Dimensional Arrays (2/2)



- Can also view 2D array as one-dimensional array

```
int sum_grades(int const *begin, int const *end) {
  int sum = 0;
  while (begin < end) {
    sum += *begin++;
  }
  return sum;
}

// call to sum_grades for student #1
int size = NUM_QUIZZES;
int sum_val = sum_grades(&grades[0][0], &grades[0][0]+size);
printf("sum_val: %d\n", sum_val);
```

# Processing Columns of Two-Dimensional Array (1/2)



- Recall `int grades[20][10];` contains grades of 20 students with each student having grades for 10 quizzes
- To compute average grade for 3rd quiz, we need to access `grades[i][2]` for all 20 rows

# Processing Columns of Two-Dimensional Array (2/2)



```
double avg_quiz_grade(int a[][NUM_QUIZZES],
                      int size, int qid) {
  int sum = a[0][qid];
  for (int i = 1; i < size; ++i) {
    sum += a[i][qid];
  }
  return (double)sum/(double)size;
}

// average of quiz #3
double avg_quiz3 =
       avg_quiz_grade(grades, NUM_STUDENTS, 2);
printf("avg grade for quiz #3: %.2f\n", avg_quiz3);
```

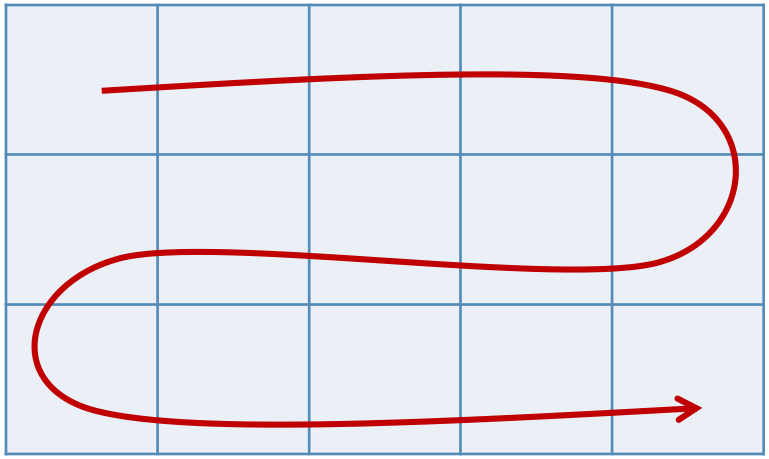
# More Computations …

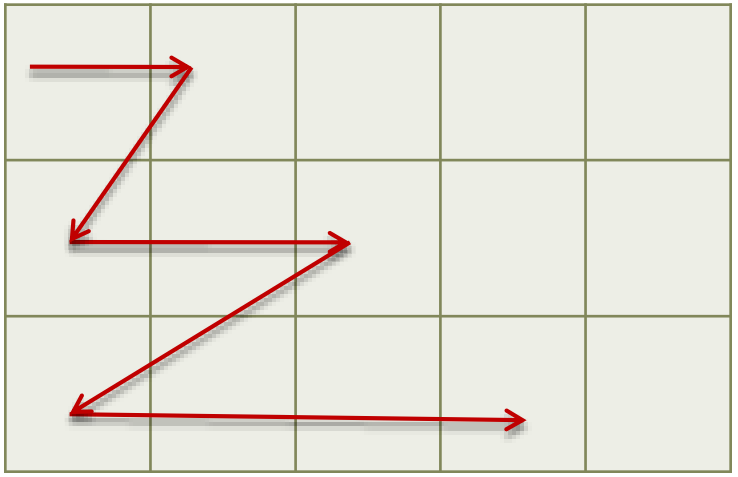


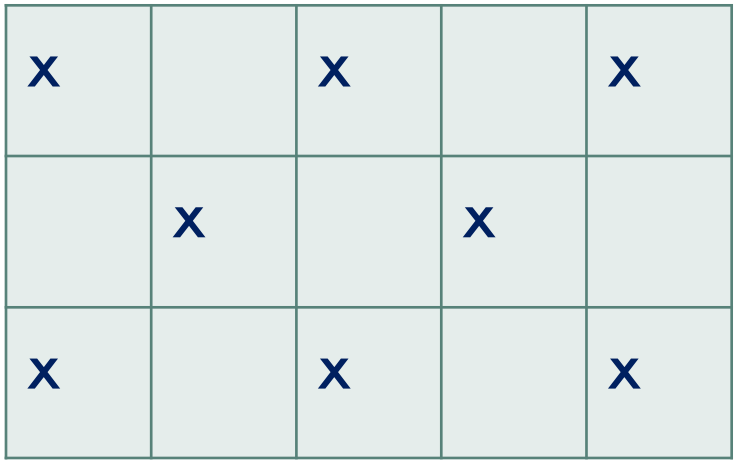
- Find maximum, minimum of all elements in 2D array
- Find a value in 2D array; how many times does value occur in array
- Compute sum of two matrices
- Swap two rows
- Swap two columns
- Find transpose of matrix
- Multiply two matrices
- Scale $M \times N$ image to size $\frac{M}{2} \times \frac{N}{2}$

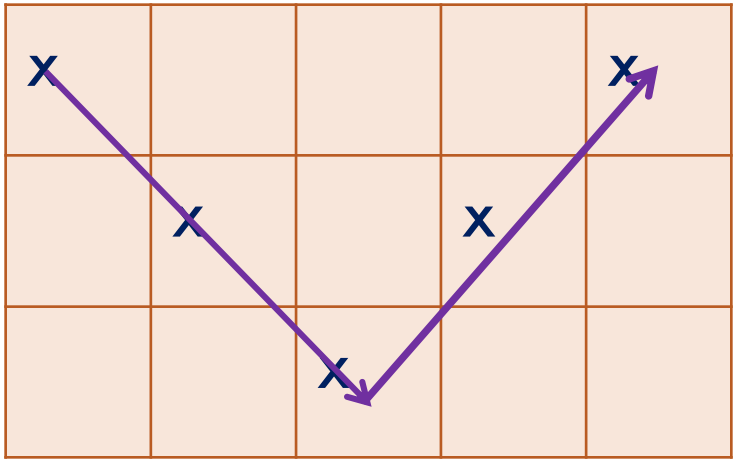
# More Traversals …









# Matrix Multiplication (1/3)



```
enum {R1=3, C1=2, R2=C1, C2=4, R3=R1, C3=C2};
double a[R1][C1], b[R2][C2], c[R3][C3];

// fill a and b with values ...
// compute c = a * b
```

| | |
|---|---|
| 3 | 4 |
| 5 | 2 |
| 1 | 6 |

x

| | | | |
|---|---|---|---|
| 2 | 3 | 7 | 1 |
| 4 | 5 | 6 | 8 |

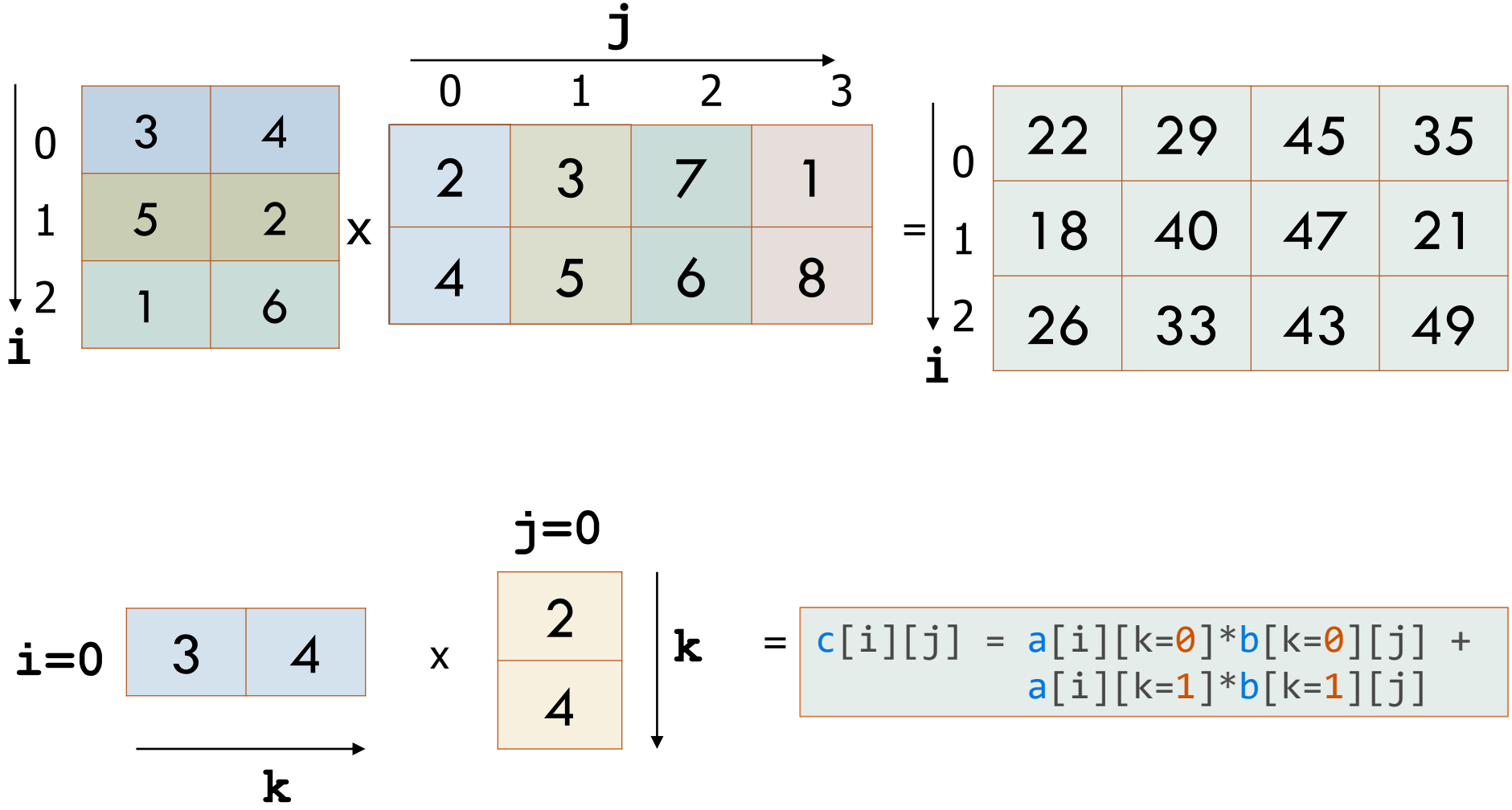
=

| | | | |
|---|---|---|---|
| 22 | 29 | 45 | 35 |
| 18 | 40 | 47 | 21 |
| 26 | 33 | 43 | 49 |

| | | | |
|---|---|---|---|
| 3*2 + 4*4=22 | 3*3 + 4*5=29 | 3*7 + 4*6=45 | 3*1 + 4*8=35 |
| 5*2 + 2*4=18 | 5*3 + 2*5=40 | 5*7 + 2*6=47 | 5*1 + 2*8=21 |
| 1*2 + 6*4=26 | 1*3 + 6*5=33 | 1*7 + 6*6=43 | 1*1 + 6*8=49 |

# Matrix Multiplication (2/3)





# Matrix Multiplication (3/3)



```
enum {R1=3, C1=2, R2=C1, C2=4, R3=R1, C3=C2};

void matrix_mul(a[][C1], int b[][C2], int c[][C3]) {
  for(int i=0; i < N; i++) {
    for(int j=0; j < L; j++) {
      c[i][j] = 0;
      for(int k=0; k < M; k++) {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

# Summary



- Two-dimensional arrays are an array of arrays
- Row-major order storage in memory
- Must specify number of columns when used as function argument