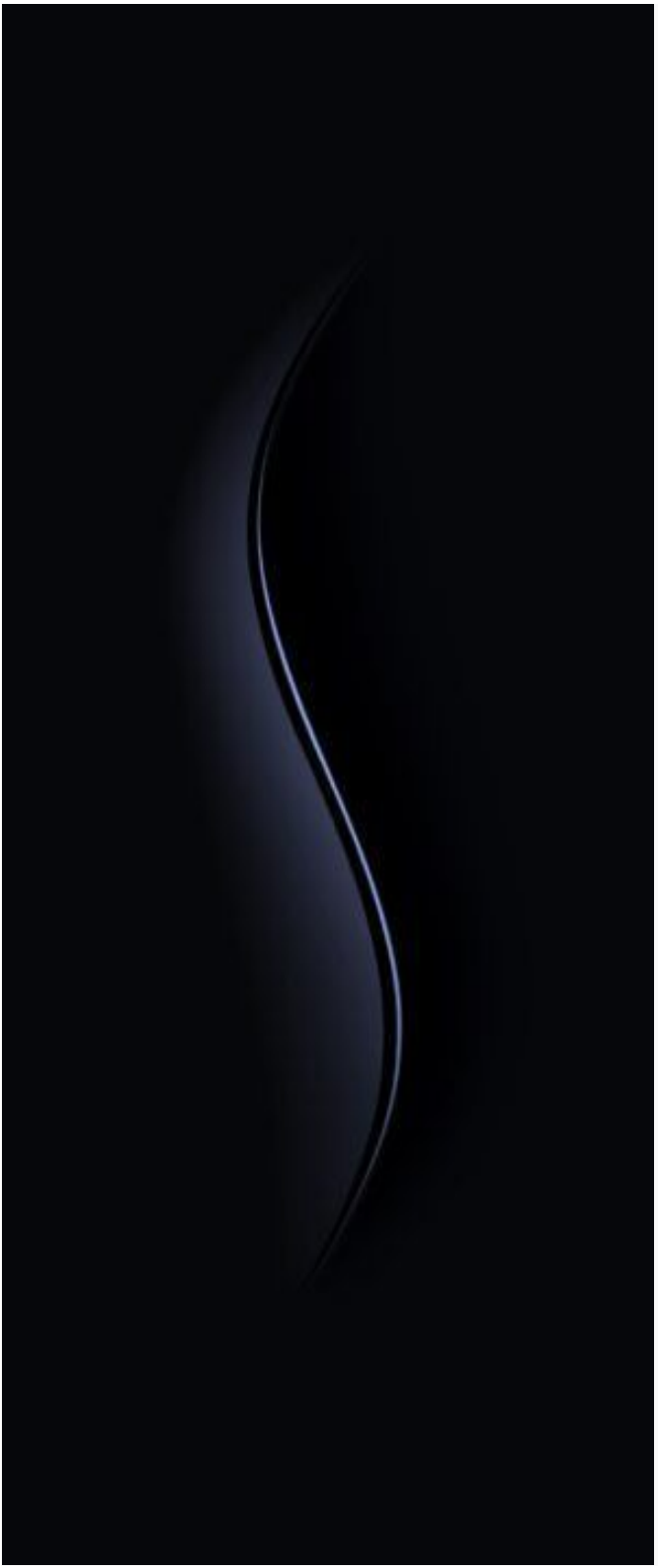


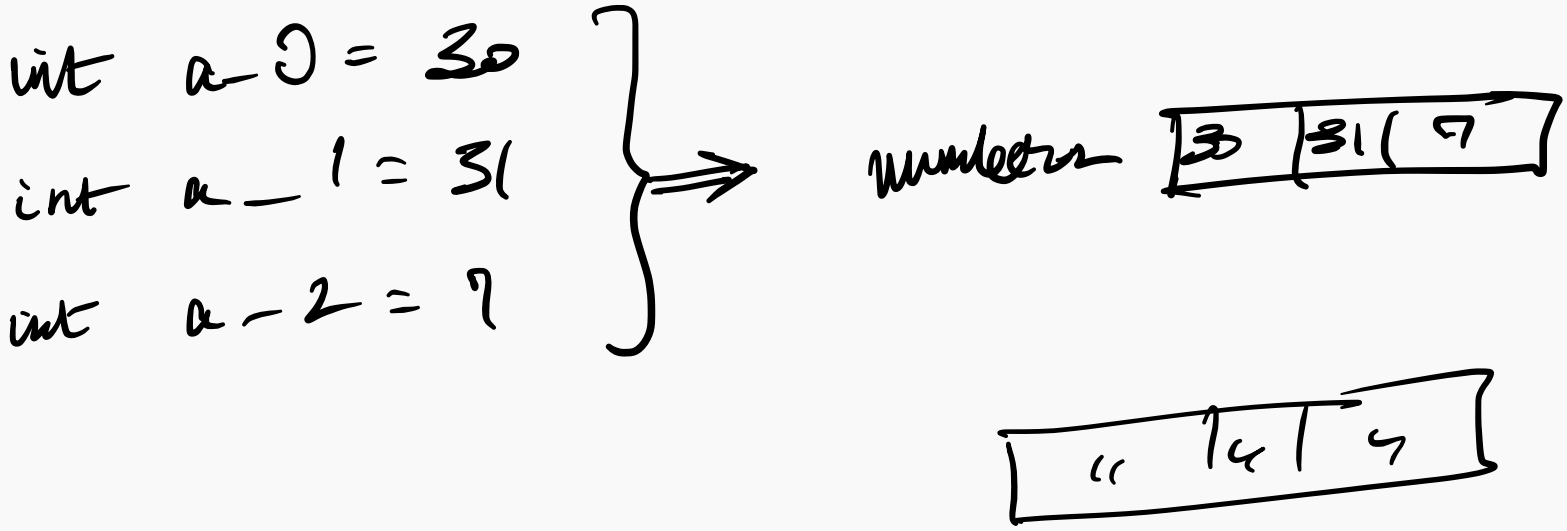
(Section No. 009)

Arrays in Many Dimensions

William T. Doan

20 November 2024

Introduction

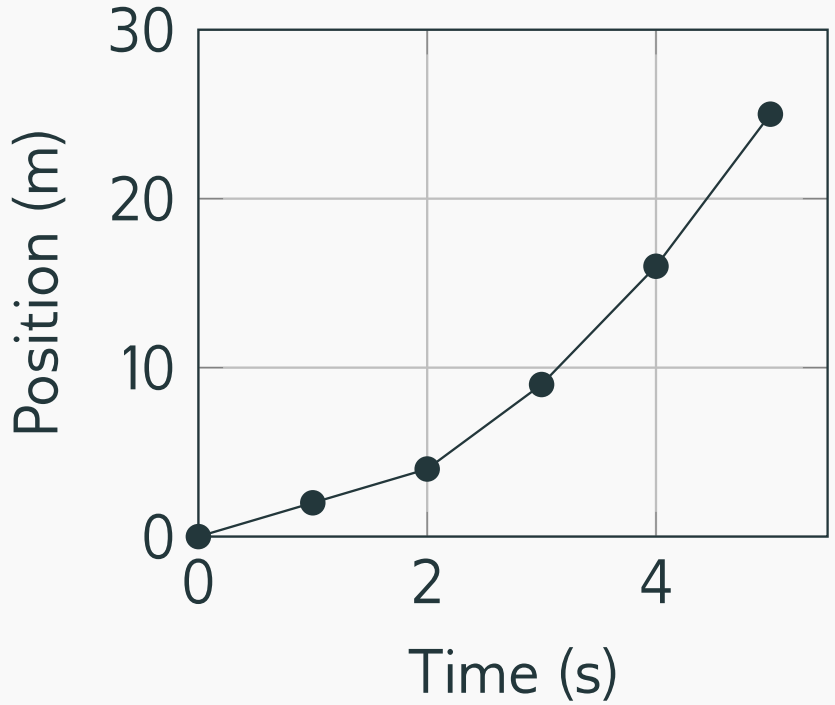


Arrays, as you have learned so far, only go in one dimension.

But data is seldom in one dimension!

Consider

Time (s)	Position (m)
0	0
1	2
2	4
3	9
4	16
5	25



Definition

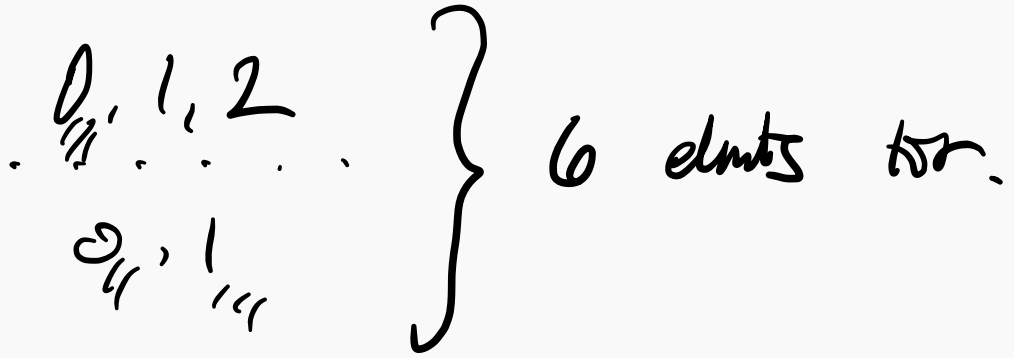
Multidimensional Arrays

A data structure which can store multiple lines of data in two or more dimensions simultaneously.

Sample No. 1

```
1
2  // An example of a 2D array.
3
4  int tests[4][3]
5
```

- We denote each successive dimension of the array with `[ ]`.
  - + **Recall:** the indices of the array are offsets which start at 0.

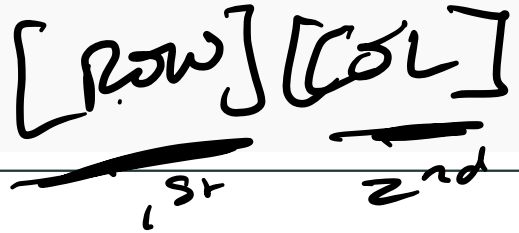


Query No. 1.

What are the valid subscripts in the first and second dimensions of the array `tests[3][2]`? How many elements does `tests` have?

2D Array Illustration

Suppose we have this code,




```
1    const int NUM_STUDENTS = 4;
2    const int NUM_TESTS = 3;
3    int tests[NUM_STUDENTS][NUM_TESTS];
4    tests[2][1] = 86;
```

The 2D array representation of such code is,

tests[0][0]	tests[0][1]	tests[0][2]
tests[1][0]	tests[1][1]	tests[1][2]
tests[2][0]	tests[2][1]	tests[2][2]
tests[3][0]	tests[3][1]	tests[3][2]

`tests[0][0]`	`tests[0][1]`	`tests[0][2]`
`tests[1][0]`	`tests[1][1]`	`tests[1][2]`
`tests[2][0]`	`tests[2][1]`	`tests[2][2]`
`tests[3][0]`	`tests[3][1]`	`tests[3][2]`

We can access an element like this,

- `tests[2][1] = 86;` means the box at row 3 column 2 is assigned the value 86.
- To access an element, use two subscripts like thus, `tests[row][column]`.

Query No. 2.

Recall: the elements of an array can be accessed using a for loop. How might we access the elements of a multidimensional array?

Nested loops!

Remarks

When we access the elements of a multidimensional array using a nested loop, one loop is used to cycle through the subscripts in each dimension.

- The subscripts of the dimension that we need to move across most quickly are cycled through in the innermost loop.
- The subscripts of the dimension that we need to move through least quickly are cycled through in the outermost loop.

Sample No. 2

```
1   const int NUM_DIVS = 3;     // Number of divisions
2   const int NUM_QTRS = 4;     // Number of quarters
3   double sales[NUM_DIVS][NUM_QTRS];
4   double totalSales = 0;      // What the array will hold
5   int div, qtr;               // Loop counters.
6
7   std::cout << "This program will calculate the total sales of\n";
8   std::cout << "all the company's divisions." << std::endl;
9   std::cout << "Enter the following sales informaton:\n\n";
10
11  // Begin nested loops to fill array.
12
13  for (div = 0; div < NUM_DIVS; div++) {
14
15      for (qtr = 0; qtr < NUM_QTRS; qtr++) {
16
17          std::cout << "Division " << (div + 1);
18          std::cout << ", Quarter " << (qtr + 1); << ": $";
19          std::cin >> sales[div][qtr]
20
21      }
22
23      std::cout << std::endl; // Print a blank line.
24
25  }
```

On List Initialization

In the same way we can initialize an array with a list, the same can be done for a 2D array.

```
1  int array[3][2] = { {5, 75}, {-9, 11}, {-20, -8} };
```

Query No. 3A.

Let us populate the table representation.

5	75
-9	11
-20	-8

Likewise, the 2D array can be partially initialized.

```
1  int array[3][2] = { {5}, {-9, 11} };
```

Query No. 3B.

Let us populate the table representation.

1.) Just 5 → all rows

5	0
-9	11
0	0

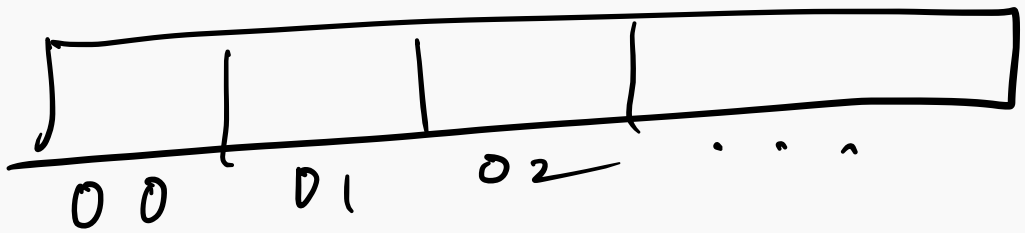
2.) Just 5 → one 5, all zeros.

On 2D Arrays in Memory

Yes, linearly!

Query No. 4.

Recall: 1D arrays are stored linearly in memory. How might 2D arrays be stored in memory?

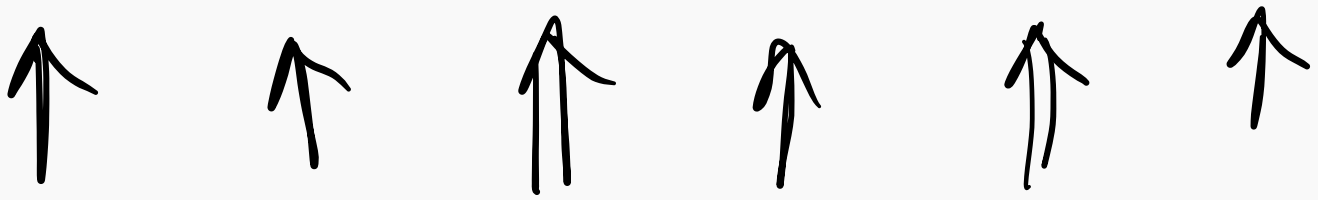


Corollary to Query No. 4.

The indices in the higher dimensions increase through all possible values before the subscripts of the lower dimensions change.

```
1  int array[3][2] = {5, 75, -9, 11};
```

Array	5	75	-9	11	0	0
Subscripts:	[0] [0]	[0] [1]	[1] [0]	[1] [1]	[2] [0]	[2] [1]



Passing 2D Arrays as Arguments

A two-dimensional array can be passed to a function as an argument.

In doing so, we must,

- use the array name in the function call.
- remember it is actually the address of the array that is passed.
- typically pass the number of elements in the first dimension in an argument as well.

The function prototype and header include one set of square brackets for each dimension.

Furthermore, the size declarator is included for every dimension, but the first. The reason for that is the array is stored linearly in memory and the compiler must know how many elements there are in higher dimensions to locate a particular element in the array.

```
1
2  // Prototype
3
4  void getScores(double [][NUMSCORES], int);
5
```

Sample No. 3

```
1   const int COLS = 4;          // Number of columns in each array
2   const int TBL1_ROWS = 3;     // Number of rows in table no. 1
3   const int TBL2_ROWS = 4;     // Number of rows in table no. 2
4
5   // Function prototype.
6
7   void showArray(const int [][COLS], int);
8
9   int main() {
10
11      int table1[TBL1_ROWS][COLS] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
12      int table2[TBL2_ROWS][COLS] = { {10, 20, 30, 40}, {50, 60, 70, 80},
13                                      {90, 100, 110, 120}, {130, 140, 15,0 160} };
14
15      std::cout << "The contents of table no. 1 are:\n";
16      showArray(table1, TBL1_ROWS);
17
18      std::cout << std::endl;
19
20      std::cout << "The contents of table no. 2 are:\n";
21      showArray(table2, TBL2_ROWS);
22
23      return 0;
24
25  }
```

```
// With arguments of a 2D array of COLS columns and the number of rows
// in an array, showArray will display the contents of the input.

void showArray(const int array[][COLS], int rows) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < COLS; j++) {

            std::cout << std::setw(4) << array[i][j] << " ";

        }

        std::cout << std::endl;

    }

}
```

Arrays in Many Dimensions

The function declaration herewith has meaning.

```
1  void showArray(const int [][COLS], int);
```

Remember. The compiler treats multidimensional arrays as a contiguous memory block, using the column size, not the row size, to access individual elements. I.e., so long as we have the number of columns, the compiler can *infer* the number of rows the array has.

Commonly, the number of rows are passed as a separate arguments. This permits a degree of flexibility in working with arrays of differing row sizes without being tied to a specific value. It is a design choice that works!

```
double scores1[4][3];
double scores2[5][3];

void getScores(double scores[][NUMSCORES], int numRows) {
    for (int i = 0; i < numRows; ++i) {
        for (int j = 0; j < NUMSCORES; ++j) {
            std::cout << "scores[" << i << "][" << j << "] = " << scores[i][j] << "\n";
        }
    }
}
```

On Arrays in Many Dimensions

We can define arrays with any number of dimensions.

```
1 short rectSolid[2][3][5];
2 double timeGrid[3][4][3][4];
```

When used as parameter, specify all but the first dimension in the prototype and function header.

```
1 void getRectSolid(short [][3][5], int);
```

Exercise No. 1

In the segment below we define an array of integers named temperatures that can store the recorded temperatures for every hour of every day for five years.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

1.) How many dimensions does temperatures have?

2.) How many elements does the array have?

3.) What are the valid subscripts in each dimension?

Exercise No. 2

Write a statement to assign the temperature 91 to the element that corresponds to the fourth year, the two hundred and eighth day, and the fifteenth hour of the array named temperatures.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

Exercise No. 3

Write a statement to display the temperature stored in the array temperatures for the last hour of the tenth day of the first year.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

Exercise Answers

1A.) three.

1B.) $5 \times 365 \times 24 = 43,800$

1C.) the first: zero to four; the second: zero to 364; the third: zero to twenty three.

The answer to Exercise No. 2 is,

```
temperatures[3][207][14] = 91;
```

The answer to Exercise No. 3 is,

```
cout << temperatures[0][9][23];
```

array_one;
array_two;
array_three[array_one, array_two]

(Let arrays 1 - 3 be 2D.)

array1[ ][ ];

array2[ ][ ];

array3[array1[2][2]][array2[3][3]]

DO NOT THIS!

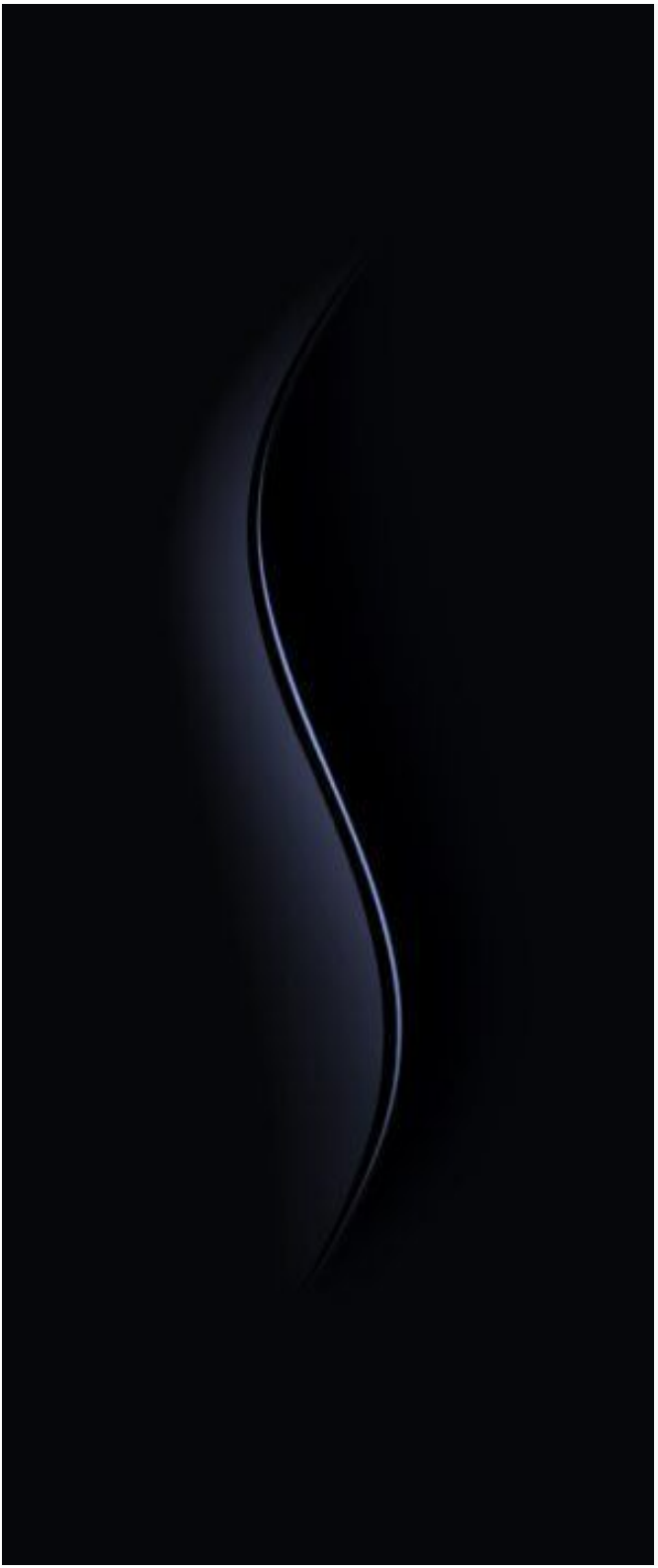
DO

4

9

if you are defining the arrays as sizes, then no.

The question gets more interesting — depending on the compiler, you can have different compilers do different things.



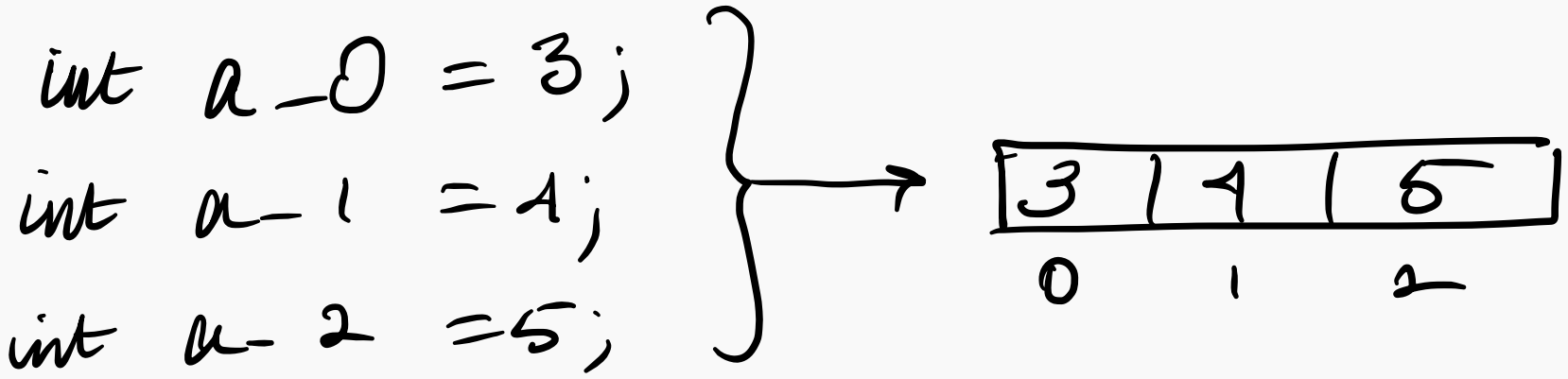
(Section No. 001)

Arrays in Many Dimensions

William T. Doan

20 November 2024

Introduction

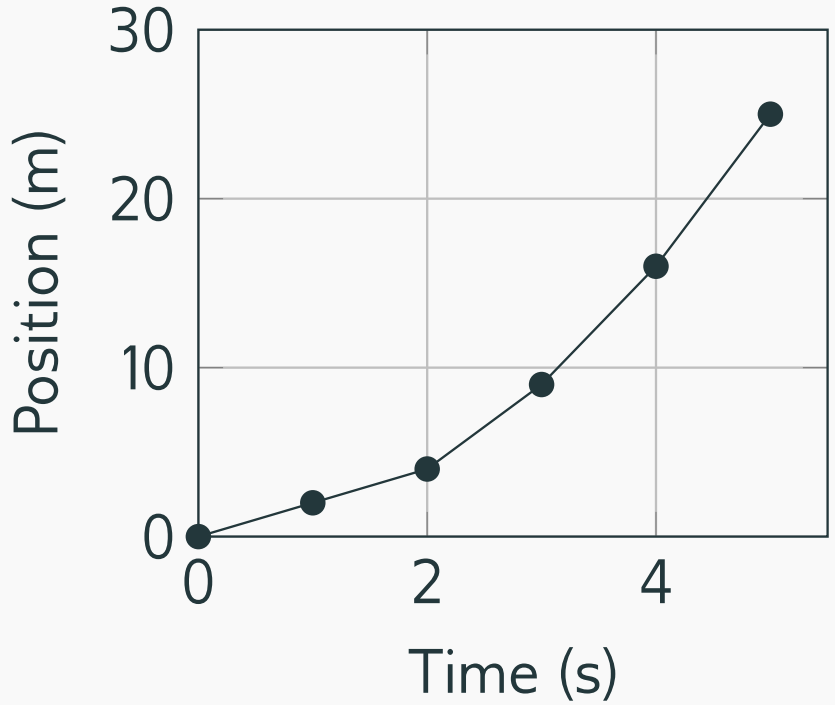


Arrays, as you have learned so far, only go in one dimension.

But data is seldom in one dimension!

Consider

Time (s)	Position (m)
0	0
1	2
2	4
3	9
4	16
5	25



Definition

Multidimensional Arrays

A data structure which can store multiple lines of data in two or more dimensions simultaneously.

Sample No. 1

```
1
2  // An example of a 2D array.
3
4  int tests[4][3]
5
```

- We denote each successive dimension of the array with `[ ]`.
  - + **Recall:** the indices of the array are offsets which start at 0.

hen: "6"

Allen: "0, 1, ..., 5"

days: "0, ..., 3"
+ hen "0, ..., 1"

Query No. 1.

What are the valid subscripts in the first and second dimensions of the array `tests[3][2]`? How many elements does `tests` have?

[Rows][Columns]

2D Array Illustration

Suppose we have this code,

```
1    const int NUM_STUDENTS = 4;
2    const int NUM_TESTS = 3;
3    int tests[NUM_STUDENTS][NUM_TESTS];
4    tests[2][1] = 86;
```

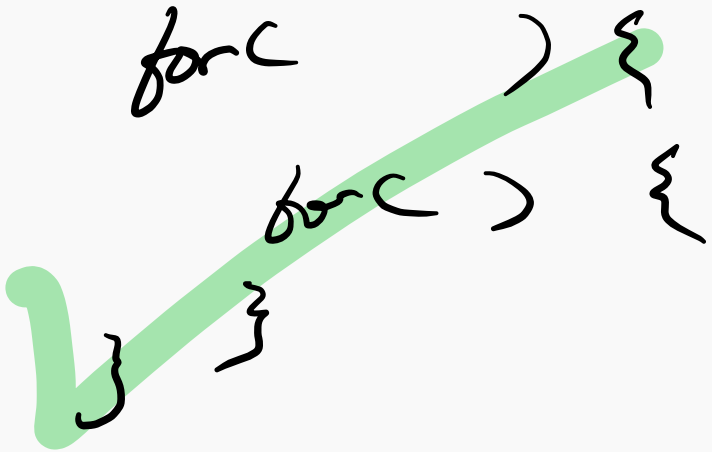
The 2D array representation of such code is,

tests[0][0]	tests[0][1]	tests[0][2]
tests[1][0]	tests[1][1]	tests[1][2]
tests[2][0]	tests[2][1]	tests[2][2]
tests[3][0]	tests[3][1]	tests[3][2]

`tests[0][0]`	`tests[0][1]`	`tests[0][2]`
`tests[1][0]`	`tests[1][1]`	`tests[1][2]`
`tests[2][0]`	`tests[2][1]`	`tests[2][2]`
`tests[3][0]`	`tests[3][1]`	`tests[3][2]`

We can access an element like this,

- `tests[2][1] = 86;` means the box at row 3 column 2 is assigned the value 86.
- To access an element, use two subscripts like thus, `tests[row][column]`.



Query No. 2.

Recall: the elements of an array can be accessed using a for loop. How might we access the elements of a multidimensional array?

Remarks

When we access the elements of a multidimensional array using a nested loop, one loop is used to cycle through the subscripts in each dimension.

- The subscripts of the dimension that we need to move across most quickly are cycled through in the innermost loop.
- The subscripts of the dimension that we need to move through least quickly are cycled through in the outermost loop.

Sample No. 2

```
1   const int NUM_DIVS = 3;     // Number of divisions
2   const int NUM_QTRS = 4;     // Number of quarters
3   double sales[NUM_DIVS][NUM_QTRS];
4   double totalSales = 0;      // What the array will hold
5   int div, qtr;               // Loop counters.
6
7   std::cout << "This program will calculate the total sales of\n";
8   std::cout << "all the company's divisions." << std::endl;
9   std::cout << "Enter the following sales informaton:\n\n";
10
11  // Begin nested loops to fill array.
12
13  for (div = 0; div < NUM_DIVS; div++) {
14
15      for (qtr = 0; qtr < NUM_QTRS; qtr++) {
16
17          std::cout << "Division " << (div + 1);
18          std::cout << ", Quarter " << (qtr + 1); << ": $";
19          std::cin >> sales[div][qtr]
20
21      }
22
23      std::cout << std::endl; // Print a blank line.
24
25  }
```

On List Initialization

In the same way we can initialize an array with a list, the same can be done for a 2D array.

```
1  int array[3][2] = { {5, 75}, {-9, 11}, {-20, -8} };
```

Query No. 3A.

Let us populate the table representation.

5	75
-9	11
-20	-8

Likewise, the 2D array can be partially initialized.

```
1  int array[3][2] = { {5}, {-9, 11} };
```

Query No. 3B.

Let us populate the table representation.

5	0
-9	11
0	0

On 2D Arrays in Memory

Still linear!

Query No. 4.

Recall: 1D arrays are stored linearly in memory. How might 2D arrays be stored in memory?

Corollary to Query No. 4.

The indices in the higher dimensions increase through all possible values before the subscripts of the lower dimensions change.

```
1  int array[3][2] = {5, 75, -9, 11};
```

Array	5	75	-9	11	0	0
Subscripts:	[0] [0]	[0] [1]	[1] [0]	[1] [1]	[2] [0]	[2] [1]

Passing 2D Arrays as Arguments

A two-dimensional array can be passed to a function as an argument.

In doing so, we must,

- use the array name in the function call.
- remember it is actually the address of the array that is passed.
- typically pass the number of elements in the first dimension in an argument as well.

The function prototype and header include one set of square brackets for each dimension.

Furthermore, the size declarator is included for every dimension, but the first. The reason for that is the array is stored linearly in memory and the compiler must know how many elements there are in higher dimensions to locate a particular element in the array.

```
1
2  // Prototype
3
4  void getScores(double [][NUMSCORES], int);
5
```

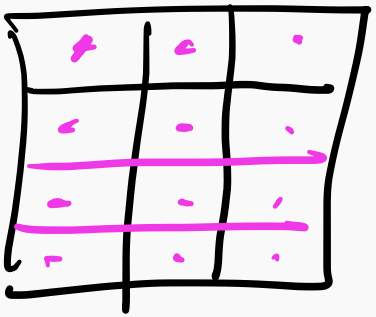
Sample No. 3

```
1   const int COLS = 4;          // Number of columns in each array
2   const int TBL1_ROWS = 3;     // Number of rows in table no. 1
3   const int TBL2_ROWS = 4;     // Number of rows in table no. 2
4
5   // Function prototype.
6
7   void showArray(const int [][COLS], int);
8
9   int main() {
10
11      int table1[TBL1_ROWS][COLS] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
12      int table2[TBL2_ROWS][COLS] = { {10, 20, 30, 40}, {50, 60, 70, 80},
13                                      {90, 100, 110, 120}, {130, 140, 15,0 160} };
14
15      std::cout << "The contents of table no. 1 are:\n";
16      showArray(table1, TBL1_ROWS);
17
18      std::cout << std::endl;
19
20      std::cout << "The contents of table no. 2 are:\n";
21      showArray(table2, TBL2_ROWS);
22
23      return 0;
24
25  }
```

```
// With arguments of a 2D array of COLS columns and the number of rows
// in an array, showArray will display the contents of the input.

void showArray(const int array[][COLS], int rows) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < COLS; j++) {

            std::cout << std::setw(4) << array[i][j] << " ";

        }

        std::cout << std::endl;

    }

}
```

Arrays in Many Dimensions

Cols := 3



The function declaration herewith has meaning.

```
1  void showArray(const int [][COLS], int);
```

Remember. The compiler treats multidimensional arrays as a contiguous memory block, using the column size, not the row size, to access individual elements. I.e., so long as we have the number of columns, the compiler can *infer* the number of rows the array has.

Commonly, the number of rows are passed as a separate arguments. This permits a degree of flexibility in working with arrays of differing row sizes without being tied to a specific value. It is a design choice that works!

```
double scores1[4][3];
double scores2[5][3];

void getScores(double scores[][NUMSCORES], int numRows) {
    for (int i = 0; i < numRows; ++i) {
        for (int j = 0; j < NUMSCORES; ++j) {
            std::cout << "scores[" << i << "][" << j << "] = " << scores[i][j] << "\n";
        }
    }
}
```

On Arrays in Many Dimensions

We can define arrays with any number of dimensions.

```
1 short rectSolid[2][3][5];
2 double timeGrid[3][4][3][4];
```

When used as parameter, specify all but the first dimension in the prototype and function header.

```
1 void getRectSolid(short [][3][5], int);
```

Exercise No. 1

In the segment below we define an array of integers named temperatures that can store the recorded temperatures for every hour of every day for five years.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

1.) How many dimensions does temperatures have?

2.) How many elements does the array have?

3.) What are the valid subscripts in each dimension?

Exercise No. 2

Write a statement to assign the temperature 91 to the element that corresponds to the fourth year, the two hundred and eighth day, and the fifteenth hour of the array named temperatures.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

Exercise No. 3

Write a statement to display the temperature stored in the array temperatures for the last hour of the tenth day of the first year.

```
1  const int NUM_YEARS = 5;
2  const int NUM_DAYS = 365;
3  const int NUM_HOURS = 24;
4  int temperatures[NUM_YEARS][NUM_DAYS][NUM_HOURS];
```

Exercise Answers

1A.) three.

1B.) $5 \times 365 \times 24 = 43,800$

1C.) the first: zero to four; the second: zero to 364; the third: zero to twenty three.

The answer to Exercise No. 2 is,

```
temperatures[3][207][14] = 91;
```

The answer to Exercise No. 3 is,

```
cout << temperatures[0][9][23];
```