Arrays

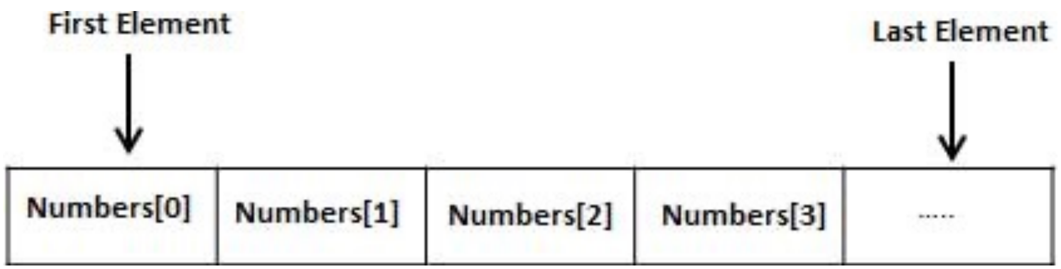
Dr Bhanu

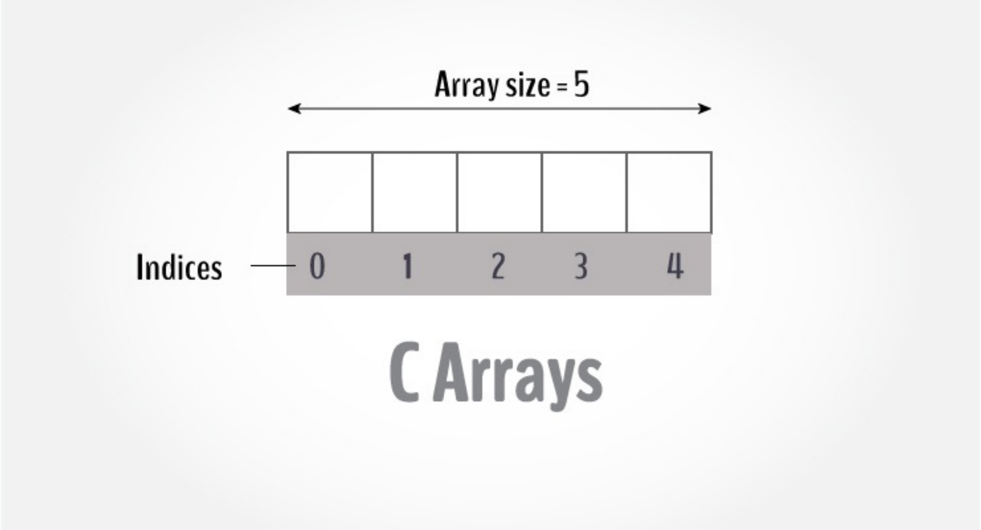
Introduction

- An **array** is a collection of data items, all of the same type, accessed using a common name.
  - A one-dimensional **array** is like a list;
  - A two dimensional **array** is like a table;
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- A specific element in an array is accessed by an index.

Elements in Array

- Array elements are stored in sequential memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.



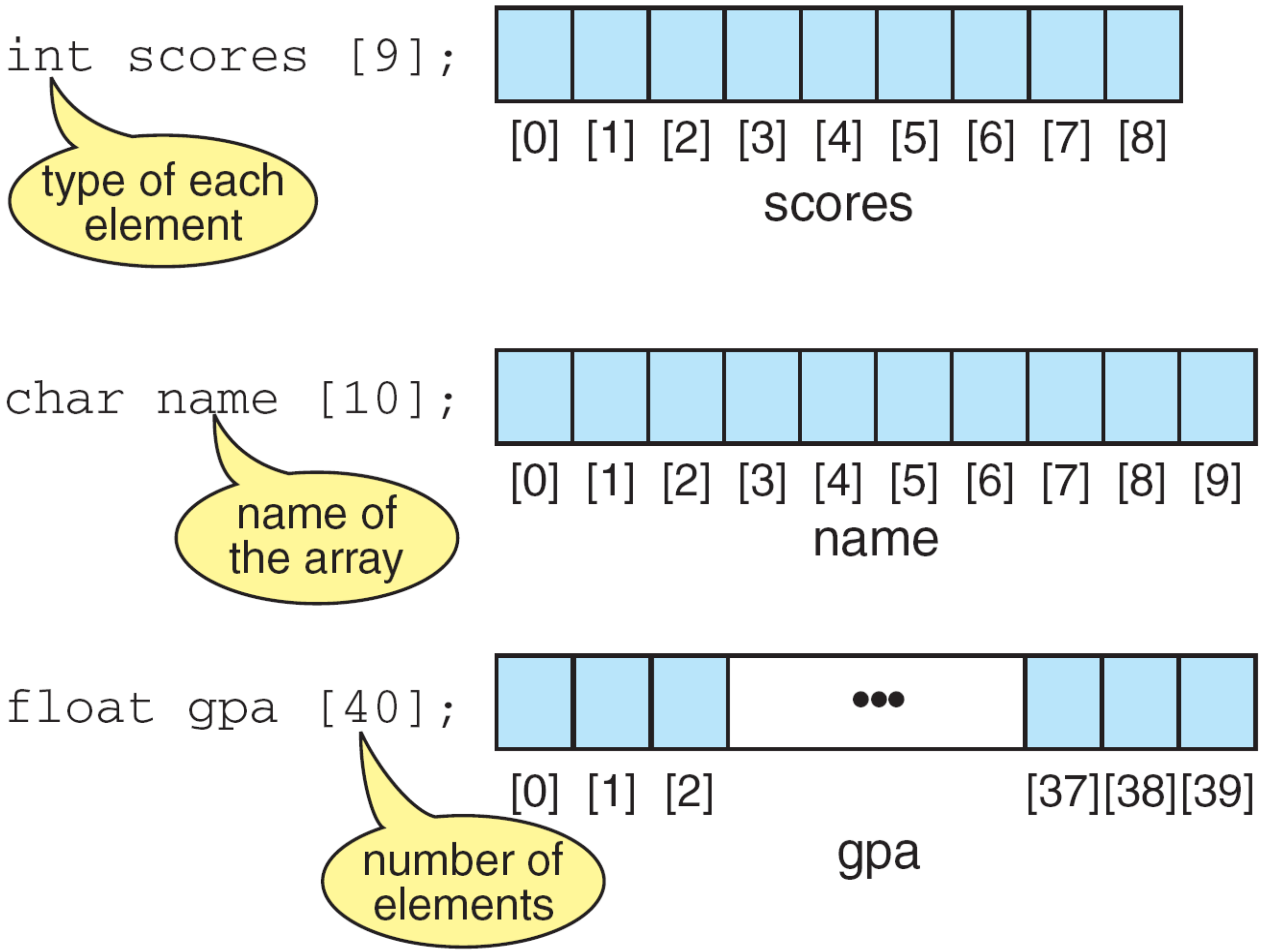


Declaring Arrays

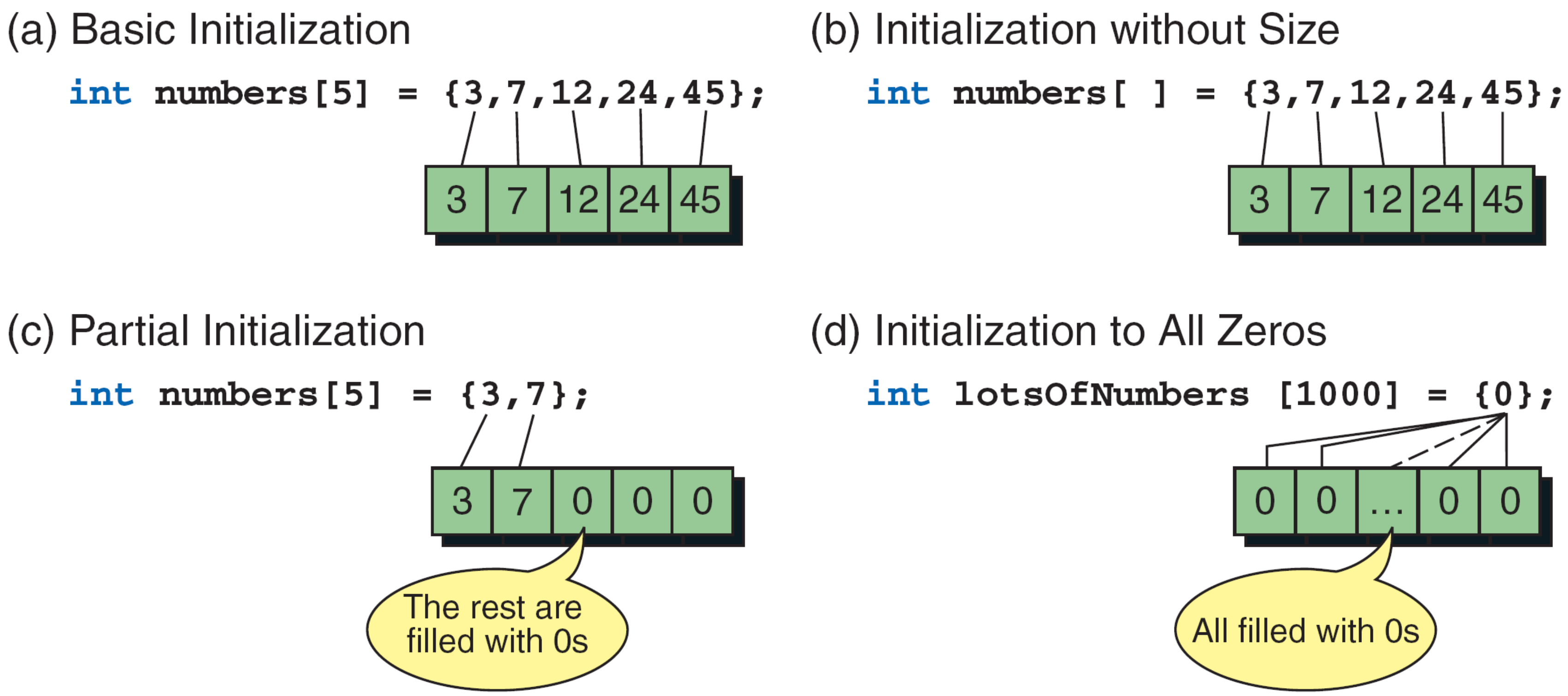
- Specify the type of the elements and the number of elements required by an array
  - type arrayName [arraySize];
  - Example : float marks[100]; double balance[10];
- The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type.
- The size and type of arrays cannot be changed (within the body of the program) after its declaration.

Examples

```
int x[10];          // An integer array named x with size 10
float GPA[30];      // An array to store the GPA for 30 students
int Scores[30][5]; // A two-dimensional array to store the scores of 5 exams for 30 students
```



Declaring and Defining Arrays



(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```

(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```

(c) Partial Initialization

```
int numbers[5] = {3,7};
```

(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```

- Only fixed-length arrays can be initialized when they are defined. Variable length arrays must be initialized by inputting or assigning the values.
- One array cannot be copied to another using assignment.

Initializing Arrays

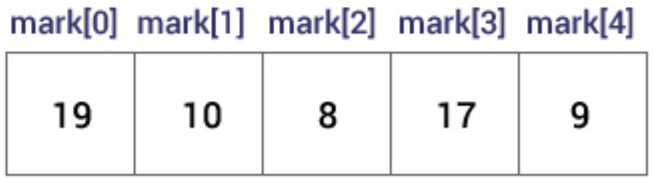
Initializing Arrays

- Initialize an array during declaration

```
int mark[5] = {19, 10, 8, 17, 9};
int mark[ ] = {19, 10, 8, 17, 9};
```

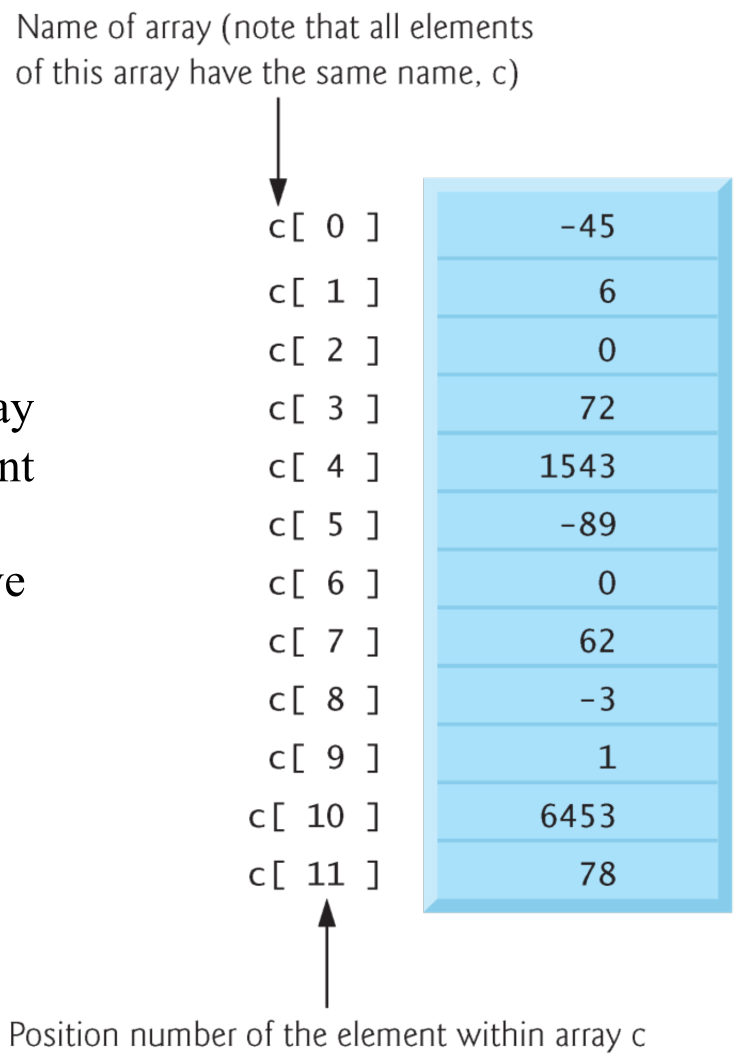
- Modify individual elements

```
mark[3] = 45;
```



mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

- Arrays occupy space in memory.
- You specify the type of each element and the number of elements required by each array so that the computer may reserve the appropriate amount of memory.
- To tell the computer to reserve 12 elements for integer array `c`, use the definition
  - `int c[ 12 ];`

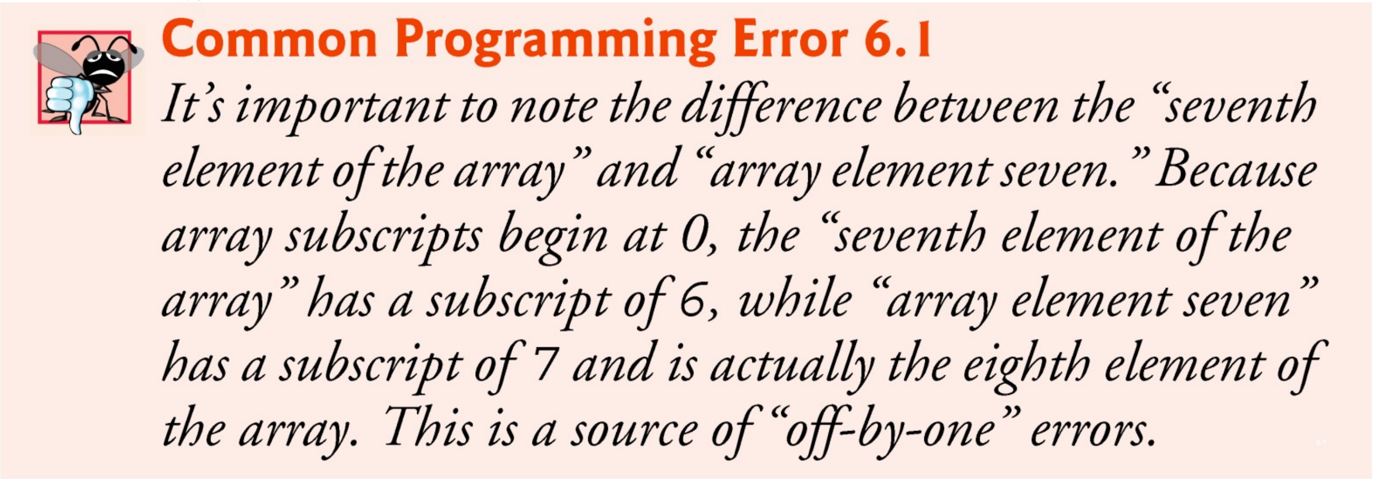


Name of array (note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

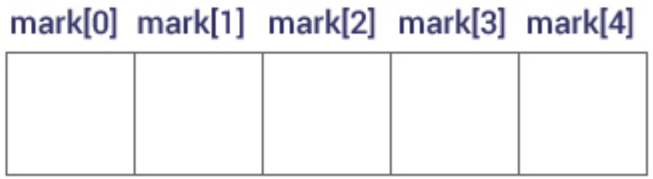
Fig. 6.1 | 12-element array.



Common Programming Error 6.1

It's important to note the difference between the "seventh element of the array" and "array element seven." Because array subscripts begin at 0, the "seventh element of the array" has a subscript of 6, while "array element seven" has a subscript of 7 and is actually the eighth element of the array. This is a source of "off-by-one" errors.

Arrays – Key points

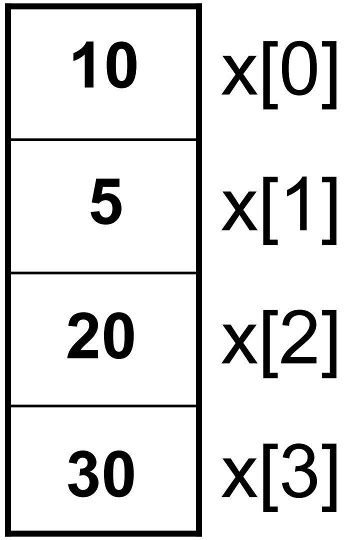


- float mark[5];
- Access individual elements of an array by indices.
- The first element is **mark[0]**, second element is **mark[1]** and so on.
- Arrays have 0 as the first index not 1.
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark[4]
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

One-dimensional Arrays

- We use one-dimensional (ID) arrays to store and access list of data values in an easy way by giving these values a common name, e.g.

```
int x[4]; // all values are named x
x[0] = 10; // the 1st value is 10
x[1] = 5;   // the 2nd value is 5
x[2] = 20; // the 3rd value is 20
x[3] = 30; // the 4th value is 30
```



10	x[0]
5	x[1]
20	x[2]
30	x[3]

Array Indices and
Out-of-bound Run-time Error

- All C one-dimensional arrays with N entries start at index 0 and end at index N-1

```
const int N=5;
int v[N]; // this array contains 5 entries
```

- It is a common error to try to access the Nth entry, e.g. v[5] or V[N], since the index of the last array entry is N-1, not N

Initializing One-dimensional Arrays

- There are two common ways to initialize one-dimensional arrays
  - Using for loop, e.g.

```
 int x[10];
for( int k = 0; k<10; k++)
           x [k] = k+1;
```

  - Specifying list of values while declaring the 1D array, e.g.

```
 int x[10] = {1,2,3,4,5,6,7,8,9,10};
 int y[ ] = {0,0,0}; // this array contains 3 entries with 0 values
double z[100] = {0}; // this array contains 100 entries, all of which are initialized to 0
double w[20] = {5,3,1}; // this array contains 20 entries the first three entries are
 initialized to 5, 3, and1 respectively while the remaining 17 entries are automatically
 initialized to 0
bool pass[10] = {true, true}; // this array contains 10 entries.
          // The first two entries are initialized to true, while the remaining 8
          // entries are automatically initialized to false
```

Storing Values in 1D Arrays

```
int x[10];
      for (int j = 0; j < 3; j++)
        scanf ("%d", &x[j]);

   for (int j = 0; j < 8; j++)
        printf ("x[%d] = %d\n", j, x[j]);
```

What do you observe?

Arrays - Observations

- It’s important to remember that arrays are not automatically initialized to zero.
- You must at least initialize the first element to zero for the remaining elements to be automatically zeroed.
- The array definition

    ```
    int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
    ```

  causes a syntax error because there are six initializers and only five array elements
- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list.
- For example,
    - int n[] = { 1, 2, 3, 4, 5 };

  would create a five-element array.

DEFINE – Symbolic Constant

```
#include <stdio.h>
# define SIZE 5

int main()
{
   int a[SIZE] = {10, -34, 23, 0, 89};
   int j;
   int sum = 0;

      for ( j = 0; j < SIZE; j++)
        sum = sum + a[j];

        printf ("sum = %d\n", sum);
   return 0;
}
```

`#define SIZE 5`

defines a symbolic constant `SIZE` whose value is `5`.
A symbolic constant is an identifier that is replaced with replacement text by the C preprocessor before the program is compiled.
Using symbolic constants to specify array sizes makes programs more scalable.

Multi-Dimensional (MD) Arrays

- C allows multidimensional arrays (arrays of arrays).

```
type name[size1][size2]...[sizeN];
```

- Example

```
int threedim[5][10][4];
```

- Simplest form is a 2-dimensional (2D) array, consisting of rows and columns (a table, a matrix etc.)

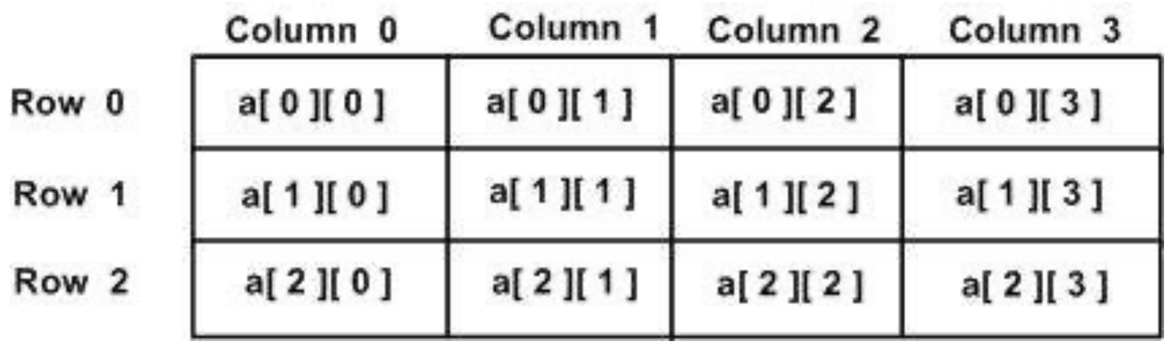
```
type arrayName [ x ][ y ];
```

- Example

```
int a[3][4];            // a table of 3 rows and 4 columns
```

2D Array

int a[3][4];



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Every element in array a is identified by an element name of the form **a[i][j]**
- “a” is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

MD Arrays - Caution

- Be careful: the amount of memory needed for an array increases exponentially with each dimension.
- For example:

```
char century [100][365][24][60][60];
```

declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!

2D Array - Initialization

```
int a[3][4] = {  {0, 1, 2, 3} , {4, 5, 6, 7} ,  {8, 9, 10, 11}};
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

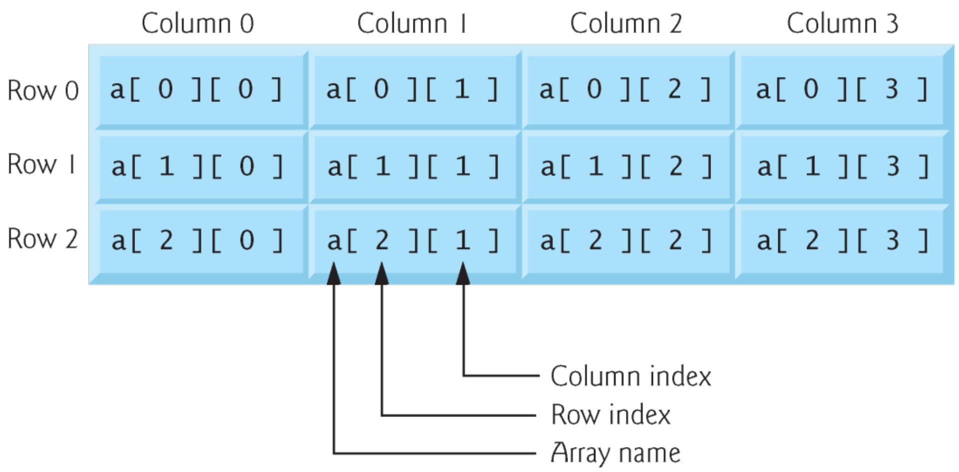
- Both the above are same.
- An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

```
Int val = a[1][3];          //Guess the value?
```

0	1	2	3
4	5	6	7
8	9	10	11

Multiple-Subscripted Arrays

- Multiple-subscripted arrays can have more than two subscripts.
- Figure 6.20 illustrates a double-subscripted array, `a`.
- The array contains three rows and four columns, so it’s said to be a 3-by-4 array.
- In general, an array with *m rows and n columns is called an m-by-n array*



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Fig. 6.20 | Double-subscripted array with three rows and four columns.

Multiple-Subscripted Arrays

- Every element in array `a` is identified in Fig. 6.20 by an element name of the form `a[i][j]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`.
- The names of the elements in the first row all have a first subscript of `0`; the names of the elements in the fourth column all have a second subscript of `3`.

Common Programming Error 6.9

Referencing a double-subscripted array element as `a[ x, y ]` *instead of* `a[ x ][ y ]`*. C interprets* `a[ x, y ]` *as* `a[ y ]`*, and as such it does not cause a compilation error.*

Multiple-Subscripted Arrays

- A multiple-subscripted array can be initialized when it's defined, much like a single-subscripted array.
- For example, a double-subscripted array `int b[2][2]` could be defined and initialized with
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
- The values are grouped by row in braces.
- The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1.
- So, the values `1` and `2` initialize elements `b[0][0]` and `b[0][1]`, respectively, and the values `3` and `4` initialize elements `b[1][0]` and `b[1][1]`, respectively.

Multiple-Subscripted Arrays

- If there are not enough initializers for a given row, the remaining elements of that row are initialized to `0`.
- Thus,
    - `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

  would initialize `b[0][0]` to `1`, `b[0][1]` to `0`, `b[1][0]` to `3` and `b[1][1]` to `4`.

How to find the Size of array

- int array[3] = {10, 20, 30};
- int array[] = {10, 20, 30};
- float a[2][2], b[2][2], result[2][2];
- char test[2][3][2];

```
#include <stdio.h>
int main()
{
  // variable
  int numbers[] = {10, 20, 30, 40, 50};

  // calculate size in bytes
  int arraySize = sizeof(numbers);
  int intSize = sizeof(numbers[0]);

  // length
  int length = arraySize / intSize;

  printf("ArraySize = %d bytes.\n", arraySize);
  printf("IntSize = %d bytes.\n", intSize);
  printf("Length of array = %d \n", length);
  return 0;
}
```

Example

```
#include <stdio.h>
int main() {
int a[5], i, search;
int pos = -1;
   printf("Enter five numbers:\n");
for (i = 0; i < 5; i++) {
      scanf("%d", &a[i]);
   }
   printf("Enter the number to search for:\n");
   scanf("%d", &search);
for (i = 0; i < 5; i++) {
if (a[i] == search) {
pos = i;
break;
      }
   }
   if (pos == -1) {
      printf("%d was not found\n", search);
   } else {
      printf("%d was found at position %d\n", search, pos);
   }
   return 0;
}
```

Multiple-Subscripted Arrays

- Figure 6.21 demonstrates defining and initializing double-subscripted arrays.
- The program defines three arrays of two rows and three columns (six elements each).
- The definition of `array1` (line 11) provides six initializers in two sublists.
- The first sublist initializes the first row (i.e., row 0) of the array to the values 1, 2 and 3; and the second sublist initializes the second row (i.e., row 1) of the array to the values 4, 5 and 6.

```c
/* Fig. 6.21: fig06_21.c
   Initializing multidimensional arrays */
#include <stdio.h>

void printArray( const int a[][ 3 ] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   /* initialize array1, array2, array3 */
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   printf( "Values in array1 by row are:\n" );
   printArray( array1 );

   printf( "Values in array2 by row are:\n" );
   printArray( array2 );

   printf( "Values in array3 by row are:\n" );
   printArray( array3 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 3.)

```c

/* function to output array with two rows and three columns */
void printArray( const int a[][ 3 ] )
{
   int i; /* row counter */
   int j; /* column counter */

   /* loop through rows */
   for ( i = 0; i <= 1; i++ ) {

      /* output column values */
      for ( j = 0; j <= 2; j++ ) {
         printf( "%d ", a[ i ][ j ] );
      } /* end inner for */

      printf( "\n" ); /* start new line of output */
   } /* end outer for */
} /* end function printArray */
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 2 of 3.)

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 3 of 3.)