# Arrays

Now, after we've learned variables and learned how to use it well, a lot of us may ask what if I wanted to save a lot of data that I will need to retrieve once more, for example:

I want to save 10 numbers, we can solve this problem easily using variables, by declaring 10 variables and save values in them.

```
int num1;
int num2;
int num3;
int num4;
int num5;
int num6;
int num7;
int num8;
int num9;
int num10;
```

Ok, that was an easy challenge, right?. But What if I needed to save 100 numbers? This can be done of course by declaring 100 variables and save data inside them, but this will be quite difficult, isn't it?

But, luckily there is a more efficient and easier way to solve this problem and it's called ***Arrays***.

## Declaration:

Let's consider the first problem above and we want to store or save 10

Numbers but using **arrays**, It will be as follows: `int num[10];`

It's quite easy right? but let's talk more about arrays and how you can use them, to declare an array you should pick the **Data type** (int, char, … etc), and a name for your array as we were doing with the **Variables**, the only new thing that we weren't doing with **Variables** is those **Parenthesis [ ]** we use them to give your **array** any size you want.

```
    DataType arrayName[arraySize];
```

## Accessing Element:

We can access elements in array by calling array name then **[**index**]** while the index refers to the index you want to access.

```
int arr[5];
arr[2] = 3;
cout << arr[2];
```

```
Output: 3
```

Arrays are zero based so the first item will be index 0 and the last item will be index of array size -1.

## Initialization:

When the array declared all its elements will contain ***Garbage*** that's why we need to initialize the array, there are many ways to initialize an array.

```
int array1[5] = {1, 2, 3, 4, 5};  // arr[0]=1,arr[1]=2...arr[4]=5.
int array2[5] = {1, 2, 3, 4, 5, 6};  // Error
int array3[5] = {1, 2, 3, 4};  // arr[0]=1,...arr[3]=4,arr[4]=0.
int array4[] = {1, 2, 3, 4, 5};  // Size of array =5.
int array5[];  // Error
```

***These ways can be used while declaring arrays only***

- In array1 the compiler will fill the array from the two **curly brackets { }** in the same order till it's end.
- If the number of element is greater the size of array as in array2 then the compiler gives an **Error.**

- If the number of element smaller than the size of array as in array3 then the compiler will fill the rest of indices with **0**.
- If you didn't mention the size of array then the compiler will set it according to the number of element initialized as in array4.
- If you didn’t mention the size of the array nor the elements in the array as in array5, the compiler will give you an **Error.**

## Assignment:

You can assign values to the elements of the array by accessing the element, and setting the value:

```
arr[2] = 3;
cin >> arr[2];
```

You can also use loops to assign values to the array elements:

```
int arr[5];
for (int i = 0; i < 5; i++)
{
    cin >> arr[i];
}
```

## Displaying :

You can display array elements by accessing the element and display its value:

```
cout << arr[2];
```

You can also use loops to display elements in the array:

```
int arr[5];
for ( int i = 0 ; i < 5 ; i++ )
{
    cout << arr[i] << ' ';
}
```

## Char Array:

char array are used to store characters.

```
char arr[5] = {'y', 'e', 's'};
```

It is preferred to use **size - 1** of the char array that's because the compiler use the last character as **NULL** character its value equal **'\0'** to detect the last element in the array, so the compiler can neglect the rest of the array

```
char arr[6] = {'a', 'b', 'c', 'd', 'e', 'f'};
cout << arr;
char arr1[6] = {'a', 'b', 'c', 'd', 'e'};
cout << arr1;
char arr2[6] = {'a', 'b', '\0', 'd', 'e'};
cout << arr2;
char arr3[6] = "abcde";
cout << arr3;
```

```
Output : abcdef5 // some problem happened due to the null character

Output : abcde  // arr1

Output : ab  // arr2 the compiler neglect the rest of array
because it reach null character.

Output : abcde // arr3
```

- Using accessing element to fill and display is better in char arrays.
  Also we can fill char array without loops.

```
char arr[10];
cin >> arr;  // abcdefg
cout << arr;
```

```
Output : abcdefg
```

# Multi-Dimensional Arrays

What if we want to represent 9*9 sudoku board?
We can create 9 arrays of size 9.

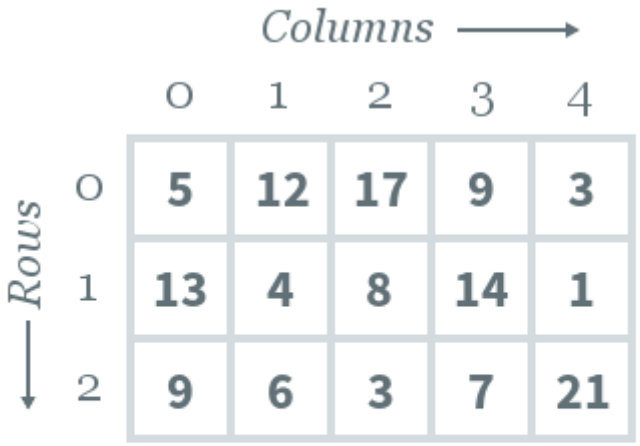
```
int row1[9];
int row2[9];
int row3[9];
int row4[9];
int row5[9];
int row6[9];
int row7[9];
int row8[9];
int row9[9];
```

Meh! too much lines.

Is this an efficient way to do so? what if we want to represent 250*250 board?
Of course not, we can easily create 2 dimensional array.

```
int Sudoku[9][9];
int Board[250][250];
```

How much easy is that like this!



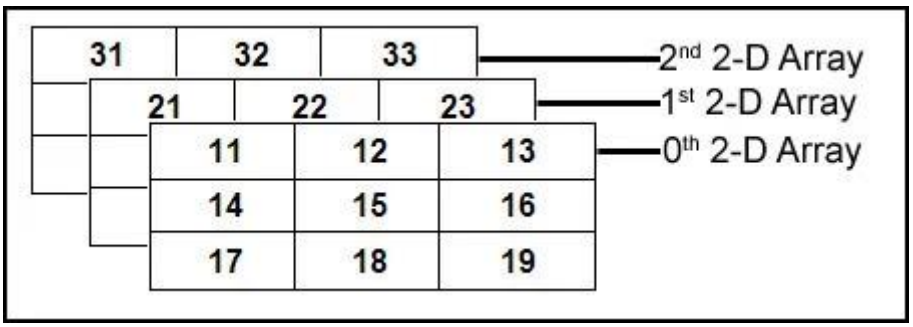

2D Array of size 3 x 5

You can make an array of any dimension(s), more dimensions in an array means more data be held.
Once you grab the logic of how the 2D array works then you can handle 3D arrays and larger.

A 3D array is an array of arrays of arrays.

```
int Table[3][3][3];
```

It can hold 27 integer type elements (3*3*3=27)




## Declaration:

To declare a 2D array you must specify the number of rows, number of columns, the data type of elements and the name of your array.

A sample form of declaration is as follows:

```
DataType ArrayName[NumberofRows][NumberofColumns];
```

To Declare a multidimensional array you must specify the number of dimensions and the size for each dimension and the data type of elements and the name of your array.

A sample form of declaration is as follows:

```
DataType ArrayName[D1Size][D2Size][D3Size].....[DnSize];
```

Where each D is a dimension, and Dn is the size of final dimension.

## Accessing Elements:

To access any element in 2D array you must specify the row index and the column index.

A sample form of accessing element is as follows :

```
ArrayName[RowIndex][ColumnIndex];
```

```
int Array[3][5]; //an array with 3 rows and 5 columns.
Array[1][2] = 8; //accessing the third element in the second row
or accessing the second element in the third column.
cout << Array[1][2]; //8
```

To access any element in a multidimensional array you must specify the index of each dimension.

A sample form of accessing element is as follows :

```
multiDimensional[D1index][D2index][D3index]....[Dnindex];
```

Where each D is a dimension, and Dn is the size of final dimension.

## Initialization of multidimensional array:

As we know, every element of a multidimensional array is an another array. So we may initialize a 2D array by specifying bracketed values for each row noting that non-given values will be automatically set to 0.

```
int Degrees[3][4] = {{30, 40, 50}, {60, 10}, {20}};
```

Also, we can just initialize it by a sequence of values noting that it won't start

initializing the upcoming row until the current one gets full. And of course, the rest of any non-given values will be set to 0. Both the first example and this one are equivalent.

```
Int Degrees[3][4] = {30, 40, 50, 0, 60, 10, 0, 0, 20, 0, 0, 0};
```

| Rows\Columns | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 30 | 40 | 50 | 0 |
| 1 | 60 | 10 | 0 | 0 |
| 2 | 20 | 0 | 0 | 0 |

initialization of any larger multi dimensional arrays will be the same way of the 2D with extra nested braces.

## Filling/ Displaying multidimensional arrays:

you may fill a 2D by setting a value for specified pairs of rows and columns.

```
int arr[3][4];
arr[0][0] = 4;//setting the value of the first element of the first row
arr[1][2] = 5;//setting the value of the third element of the second row
```

To display existed values:

```
cout << arr[0][0];
cout << arr[1][2];
```

Output:

```
4
5
```

Also, you may fill the whole array by iterating over every element.

```
int arr[3][4];
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
    {
        arr[i][j] = i * j;
    }
}
```

And for displaying the whole array:

```
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
    {
        cout << arr[i][j] << ' ';
    }
    cout << endl;
}
```

Output:

```
0 0 0 0
0 1 2 3
0 2 4 6
```

Filling/ Displaying any larger multidimensional array will be the same way with extra nested brackets/ loops.