Arrays

Suppose you have a long list of numbers, but you don't want to assign them to variables individually.

For example, you are writing a simple program for a restaurant to keep a list of the amount each diner has, but you don't want to write a list like:

float joe, betty, charls…;

Joe = 88.33;

Betty = 17.23;

charls = 55.55; etc.

A list like that could run to hundreds or thousands of entries.

This is why arrays were invented:

Arrays are a convenient way to group many variables under a single variable name.

Arrays can be:

- one-dimensional like a list
- two-dimensional like a chessboard
- three-dimensional like an apartment building
- or can have any arbitrary dimensionality, including ones humans can't visualise easily.

An array is defined using square brackets [ ]

Declaration of array: array of 3 integers called my_array would be declared as:

int my_array[3]; *// no space between name of array and '['.*

- **This statement would cause space for three adjacent integers to be created in memory as:**

my_array: 

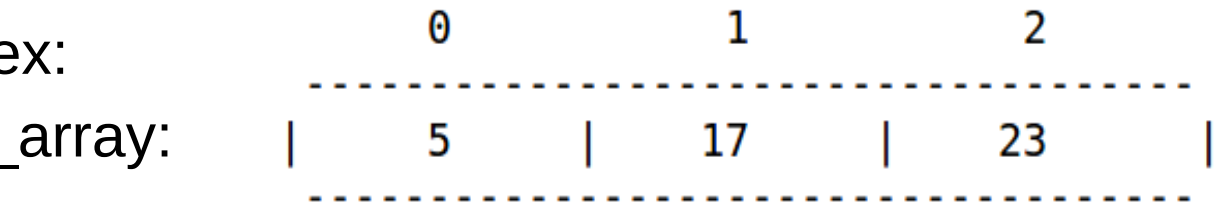
- **Three integer in the above array are called its locations**
- **Values filling them are called the array's elements.**
- **The position of an element in the array is called its index** (the plural is indices).

Arrays in C are zero-based, so the indices of array run (0, 1, 2)

int my_array[3];
my_array[0] = 5;
my_array[1] = 17;
my_array[2] = 23;

In this example, 5, 17, and 23 are the array's elements, and 0, 1, and 2 are its corresponding indices.

The above example would result in an array like:



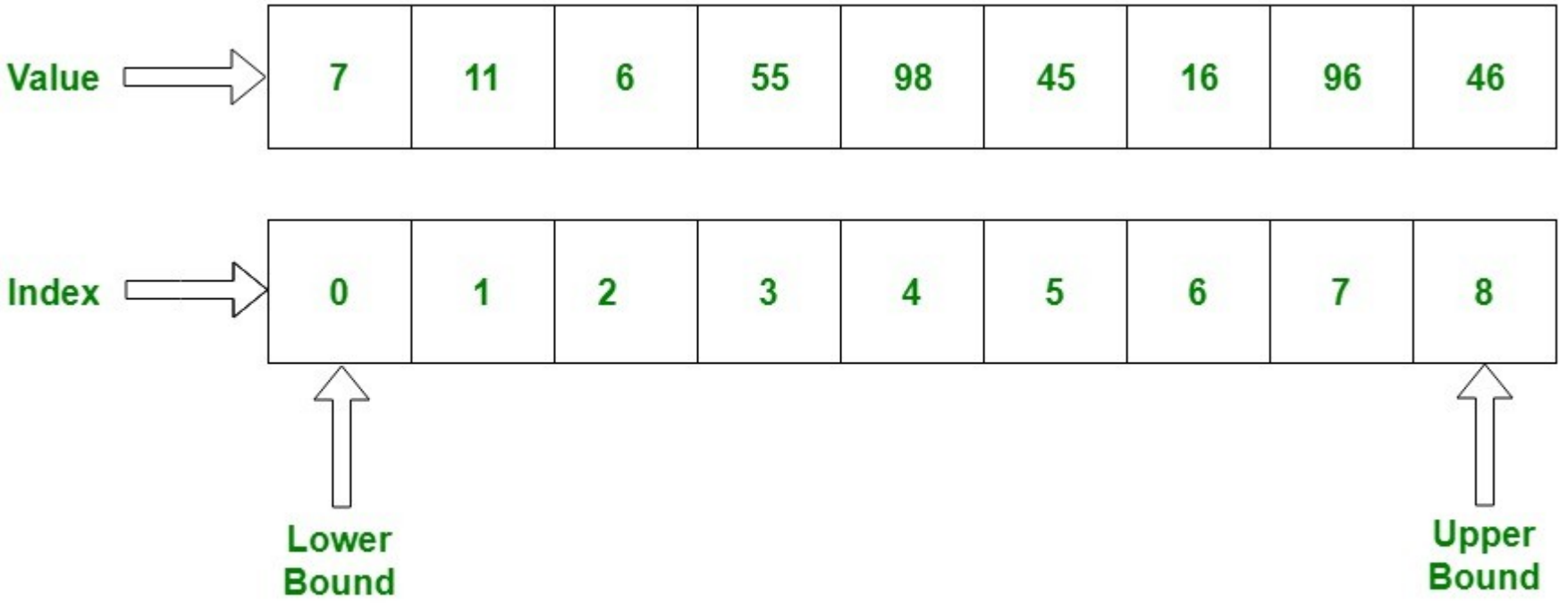
Note that **every element in an array must be of the same type**, for example, integer. It is not possible in C to have arrays that contain multiple data types.

Array bounds

The code below defines an array with 9 elements (C arrays are zero-based.) : ***char my_array[9];***

The first and last positions in an array are called its bounds. Remember that the bounds of an array are zero and the integer that equals the number of elements it contains, minus one.

In C's, the compiler does not complain if you try to write to elements of an array that do not exist(i.e. if you exceed the bounds of an array), the debugger can tell you at run-time.



Arrays and for loops

When you declare an array, the computer allocates a block of memory for it, but the **alocated block contains garbage** (**random values**).

So, **before using an array, you should initialise it**. **The easiest way to initialise an array is with a for loop.**

- It is usually a good idea to set all elements in the array to zero.

  my_array[index] = 0;

- You can also fill the array with different values.

  my_array[index] = index;

  this fills each element of the array with its own index:

```c
//C program to set all elements in the array to zero

#include <stdio.h>
int main ()
{
 int index, my_array[5];

 for (index = 0; index < 5; index++)
  {
   my_array[index] = 0;
   printf ("my_array[%d] = %d \n", index, my_array[index]);
  }

 return 0;
}
```

Output:

```
my_array[0] = 0
my_array[1] = 0
my_array[2] = 0
my_array[3] = 0
my_array[4] = 0
```

Multidimensional arrays

These are **Arrays with more than one dimension.**

Suppose that you are writing a chess-playing program. A chessboard is an 8-by-8 grid. What data structure would you use to represent it?

You could use an array that has a chessboard-like structure, that is, a two-dimensional array, to store the positions of the chess pieces.

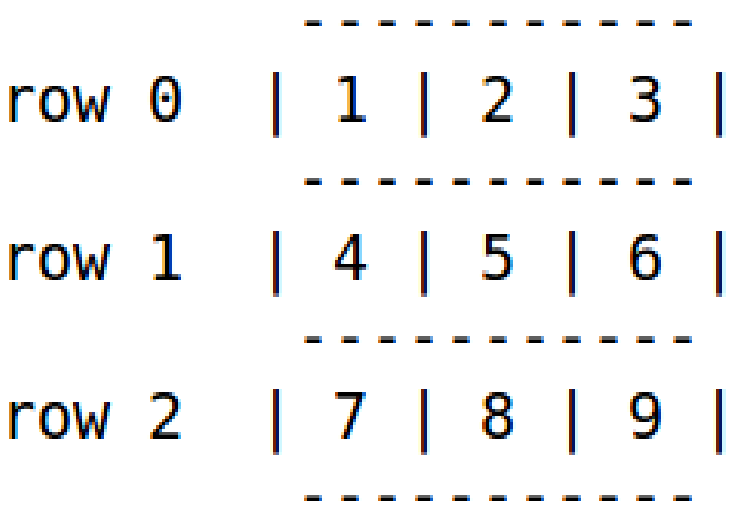
You can declare an array of two dimensions as follows:

variable_type array_name[size1][size2]

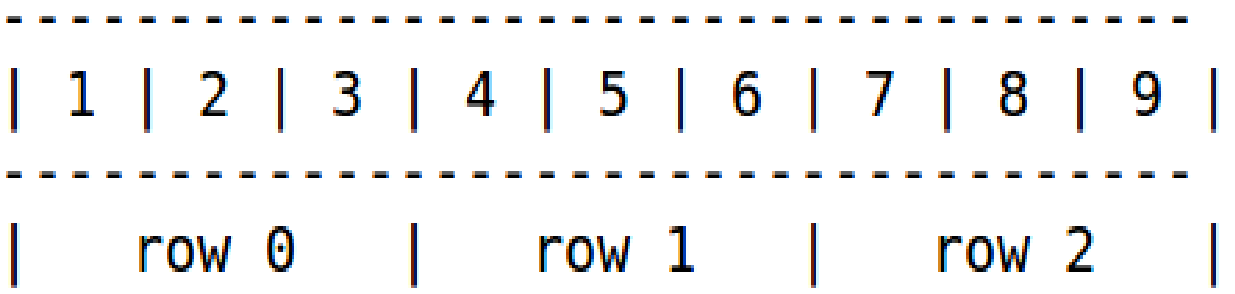
Example: ***int chessboard[8][8];***

Two-dimensional arrays use two indices to pinpoint an individual element of the array.

Consider the following 2D array. **Every element in this grid needs two indices to pin-point it.** The first index gives the row number for the grid and the second index gives the column number.



Since computer memory is essentially one-dimensional, with memory locations running straight from 0 up through the highest location in memory, a multidimensional array cannot be stored in memory as a grid. Instead, the array is dissected and stored in rows.



Arrays are stored by row:

A two-dimensional array is always thought of as follows:

array_name[row][column]

Every row stored will contain elements of many columns. The column index runs from 0 to size - 1 inside every row in the one-dimensional representation (where size is the number of columns in the array), **so the column index is changing faster than the row index,** as the one-dimensional representation of the array inside the computer is traversed.

You can represent a three-dimensional array, such as a cube, in a similar way:

variable_type array_name[size1][size2][size3]

3D arrays are stored in the same basic way as 2D, They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest (then the next-to-last, and so on).

Arrays and nested loops

To initialize multidimensional arrays, you can use nested for loops.

The last index is always the one that varies fastest (then the next-to-last, and so on).

In the example output, you can see that the column index changes every line, while the row index changes every three lines.

C Program To initialize 2D array using for loop:

```c
#include <stdio.h>
#define RowSize 3
#define ColumnSize 3

int main ()
{
 int row, column;
 int my_array[RowSize][ColumnSize];

 for (row = 0; row < RowSize; row++)
  {
   for (column = 0; column < ColumnSize; column++)
    {
        my_array[row][column] = 0;
        printf("my_array[%d][%d] DONE\n", row, column);
    }
  }
 printf("\n");
}
```

Output:

```
my_array[0][0] DONE
my_array[0][1] DONE
my_array[0][2] DONE
my_array[1][0] DONE
my_array[1][1] DONE
my_array[1][2] DONE
my_array[2][0] DONE
my_array[2][1] DONE
my_array[2][2] DONE
```

C Program To initialize 3D array using for loop:

```c
#include <stdio.h>
#define SIZE1 3
#define SIZE2 3
#define SIZE3 3

int main ()
{
 int i, j, k;
 int my_array[SIZE1][SIZE2][SIZE3];

 for (i = 0; i < SIZE1; i++)
  {
   for (j = 0; j < SIZE2; j++)
    {
     for (k = 0; k < SIZE3; k++)
      {
       my_array[i][j][k] = 0;
       printf("my_array[%d][%d][%d] DONE\n", i, j, k);
      }
    }
  }
 printf("\n");
}
```

Output:

```
my_array[0][0][0] DONE
my_array[0][0][1] DONE
my_array[0][0][2] DONE
my_array[0][1][0] DONE
my_array[0][1][1] DONE
my_array[0][1][2] DONE
my_array[0][2][0] DONE
my_array[0][2][1] DONE
my_array[0][2][2] DONE
my_array[1][0][0] DONE
my_array[1][0][1] DONE
my_array[1][0][2] DONE
my_array[1][1][0] DONE
my_array[1][1][1] DONE
my_array[1][1][2] DONE
my_array[1][2][0] DONE
my_array[1][2][1] DONE
my_array[1][2][2] DONE
my_array[2][0][0] DONE
my_array[2][0][1] DONE
my_array[2][0][2] DONE
my_array[2][1][0] DONE
my_array[2][1][1] DONE
my_array[2][1][2] DONE
my_array[2][2][0] DONE
my_array[2][2][1] DONE
my_array[2][2][2] DONE
```

Initializing arrays

You must initialize your arrays or they will contain garbage. There are two main ways:

1) By assigning values to array elements individually or with for loops:

my_array[0] = 42;

my_array[1] = 52;

my_array[2] = 23;

2) The second method is more efficient and less tedious. It uses a single assignment operator (=) and a few curly brackets ({…}).

A 3 by 3 array could be initialized in the following way:

```
int my_array[3][3] = {   {1, 2, 4 },
                         {1, 6, 0},
                         {4, 2, 0}  };
```

The same array initialization could be written this way:

```
int my_array[3][3] =  {1, 2, 4, 1, 6, 0, 4, 2, 0};
```

```c
//C program to initialize 1D arrays in different ways:
#include <stdio.h>
int main()
{
    int i;
    int arr[4];
    /*-------------------------------------------------------------------------
    Method 1: Assigning values to array elements individually
            arr[0] = 5;
            arr[1] = 7;
            arr[2] = -10;
            arr[3] = arr[0];
    //-------------------------------------------------------------------------
    Method 2: assigning values to array elements with for loops:

            for(i=0; i<4; i++)
            {   printf("Enter element %d:", i);
                scanf("%d",&arr[i]);
            }
    */ /*-------------------------------------------------------------------------
    Method 3: uses a single assignment operator (=) and a few curly brackets ({…})

            int arr[4] = { 10, 20, 30, 40 }
    *///-------------------------------------------------------------------------

    printf("%d %d %d %d\n", arr[0], arr[1], arr[2], arr[3]);
    return 0;
}
```

```c
//C program for 2D array initialization
#include <stdio.h>
int main()
{
 int row, column;
 int my_array[3][3] =
 {                              // array initialization could be written this way:
  {1, 2, 4},                    // int my_array[3][3] =  {1,2,4,1,6,0,4,2,0};
  {1, 6, 0},
  {4, 2, 0}
 };

 for (row = 0; row <=2; row++)
  {
   for (column = 0; column <= 2; column++)
    {
     //printf("%d\t", my_array[row][column]);
     printf("my_array[%d][%d]=%d\n", row, column, my_array[row][column] );
    }
   printf("\n");
  }
 return 0;
}
```