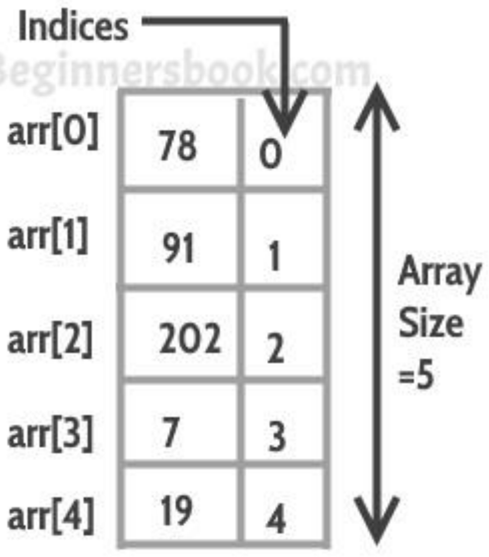
Arrays

An array is a collection of similar items stored in contiguous memory locations. In programming, sometimes a simple variable is not enough to hold all the data. For example, lets say we want to store the marks of 500 students, having 500 different variables for this task is not feasible, we can define an array with size 500 that can hold the marks of all students.



Declaring an array in C++

There are couple of ways to declare an array.

Method 1:

```
int arr[5];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

Method 2:

```
int arr[] = {10, 20, 30, 40, 50};
```

Method 3:

```
int arr[5] = {10, 20, 30, 40, 50};
```

Accessing Array Elements

Array index starts with 0, which means the first array element is at index 0, second is at index 1 and so on. We can use this information to display the array elements. See the code below:

```
#include <iostream>
using namespace std;

int main(){
   int arr[] = {11, 22, 33, 44, 55};
   cout<<arr[0]<<endl;
   cout<<arr[1]<<endl;
```

```
  cout<<arr[2]<<endl;
  cout<<arr[3]<<endl;
  cout<<arr[4]<<endl;
  return 0;
}
```

Output:

```
11
22
33
44
55
```

Although this code worked fine, displaying all the elements of array like this is not recommended. When you want to access a particular array element then this is fine but if you want to display all the elements then you should use a loop like this:

```
#include <iostream>
using namespace std;

int main(){
  int arr[] = {11, 22, 33, 44, 55};
  int n=0;

  while(n<=4){
    cout<<arr[n]<<endl;
    n++;
  }
  return 0;
}
```

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of three integers, you could write:

```
int myNum[3] = {10, 20, 30};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the **first element** in **cars**:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];
// Now outputs Opel instead of Volvo
```

Loop Through an Array

You can loop through the array elements with the for loop.

The following example outputs all elements in the **cars** array:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
for(int i = 0; i < 4; i++) {
  cout << cars[i] << "\n";
}
```

Omit Array Size

You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it:

```
string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3
```

This is completely fine. However, the problem arise if you want extra space for future elements. Then you have to overwrite the existing values:

~~string cars[] = {"Volvo", "BMW", "Ford"};~~

```
string cars[] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
```

If you specify the size however, the array will reserve the extra space:

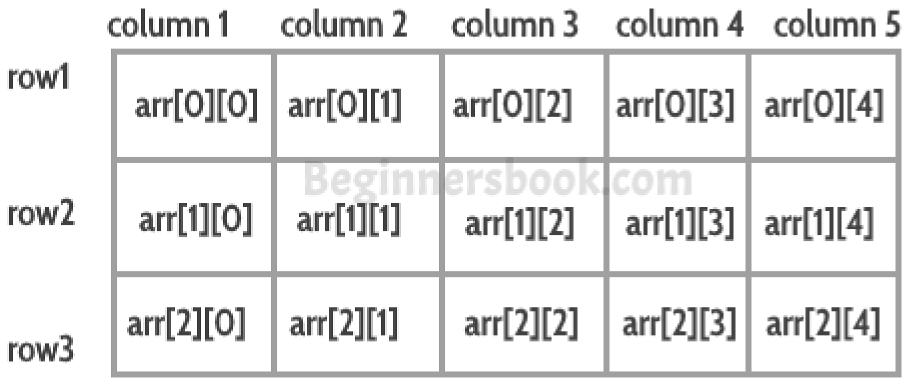
```
string cars[5] = {"Volvo", "BMW", "Ford"}; // size of array is 5, even though it's only three elements inside it
```

Now you can add a fourth and fifth element without overwriting the others:

```
string cars[3] = {"Mazda"};
string cars[4] = {"Tesla"};
```

Multidimensional Arrays

Multidimensional arrays are also known as **array of arrays**. The data in multidimensional array is stored in a tabular form as shown in the diagram below:



	column 1	column 2	column 3	column 4	column 5
row1	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
row2	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
row3	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]

A two dimensional array:
int arr[2][3];
This array has total 2*3 = 6 elements.
A three dimensional array:
int arr[2][2][2];
This array has total 2*2*2 = 8 elements.
Two dimensional array
Lets see how to declare, initialize and access Two Dimensional Array elements.
How to declare a two dimensional array?
int myarray[2][3];
Initialization:
We can initialize the array in many ways:
Method 1:
int arr[2][3] = {10, 11 ,12 ,20 ,21 , 22};
Method 2:
This way of initializing is preferred as you can visualize the rows and columns here.
int arr[2][3] = {{10, 11 ,12} , {20 ,21 , 22}};
Accessing array elements:
arr[0][0] – first element
arr[0][1] – second element
arr[0][2] – third element
arr[1][0] – fourth element
arr[1][1] – fifth element
arr[1][2] – sixth element
Example: Two dimensional array in C++

```
#include <iostream>
using namespace std;

int main(){
  int arr[2][3] = {{11, 22, 33}, {44, 55, 66}};
  for(int i=0; i<2;i++){
    for(int j=0; j<3; j++){
      cout<<"arr["<<i<<"]["<<j<<"]: "<<arr[i][j]<<endl;
    }
  }
  return 0;
}
```

Output:

```
arr[0][0]: 11
arr[0][1]: 22
arr[0][2]: 33
arr[1][0]: 44
arr[1][1]: 55
arr[1][2]: 66
```

Advantages of an Array in C/C++:

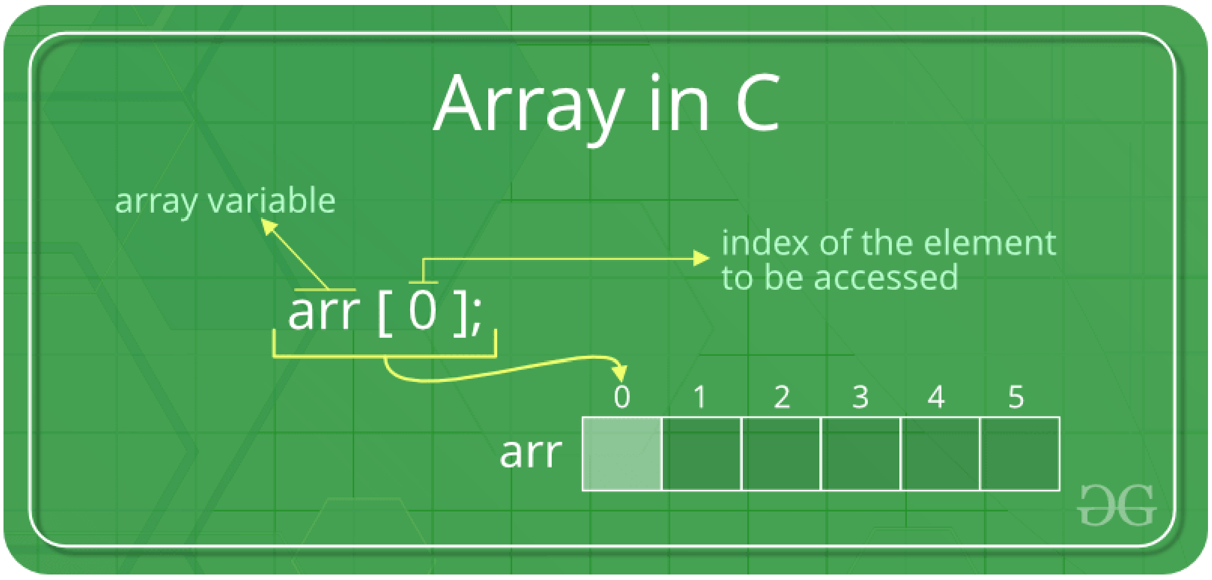
1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Facts about Array in C/C++:

- **Accessing Array Elements:**
  Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



Example:

```
#include <iostream>
using namespace std;

int main()
{
   int arr[5];
   arr[0] = 5;
   arr[2] = -10;

   // this is same as arr[1] = 2
   arr[3 / 2] = 2;
   arr[3] = arr[0];

   cout << arr[0] << " " << arr[1]
       << " " << arr[2] << " " << arr[3];

   return 0;
}
```

Output:

```
5 2 -10 5
```

No Index Out of bound Checking:

There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
// This C++ program compiles fine
// as index out of bound
// is not checked in C.

#include <iostream>
```

```
using namespace std;

int main()
{
    int arr[2];

    cout << arr[3] << " ";
    cout << arr[-2] << " ";

    return 0;
}
```

Output:

```
2008101287 4195777
```

In C, it is not compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just Warning.

```
#include <stdio.h>
int main()
{

    // Array declaration by initializing it with more
    // elements than specified size.
    int arr[2] = { 10, 20, 30, 40, 50 };

    return 0;
}
```

 Warnings:

```
prog.c: In function 'main':
prog.c:7:25: warning: excess elements in array initializer
  int arr[2] = { 10, 20, 30, 40, 50 };
                         ^
prog.c:7:25: note: (near initialization for 'arr')
prog.c:7:29: warning: excess elements in array initializer
  int arr[2] = { 10, 20, 30, 40, 50 };
                             ^
prog.c:7:29: note: (near initialization for 'arr')
prog.c:7:33: warning: excess elements in array initializer
  int arr[2] = { 10, 20, 30, 40, 50 };
                                 ^
prog.c:7:33: note: (near initialization for 'arr')
```

Note: The program won’t compile in C++. If we save the above program as a .cpp, the program generates compiler error *“error: too many initializers for ‘int [2]'”*.

 The elements are stored at contiguous memory locations
Example:

```
// C++ program to demonstrate that array elements
// are stored contiguous locations

#include <iostream>
using namespace std;

int main()
{
   // an array of 10 integers.  If arr[0] is stored at
   // address x, then arr[1] is stored at x + sizeof(int)
   // arr[2] is stored at x + sizeof(int) + sizeof(int)
   // and so on.
   int arr[5], i;

   cout << "Size of integer in this compiler is " << sizeof(int) << "\n";

   for (i = 0; i < 5; i++)
      // The use of '&' before a variable name, yields
      // address of variable.
      cout << "Address arr[" << i << "] is " << &arr[i] << "\n";

   return 0;
}
```

Output:

```
Size of integer in this compiler is 4
Address arr[0] is 0x7ffd636b4260
Address arr[1] is 0x7ffd636b4264
Address arr[2] is 0x7ffd636b4268
Address arr[3] is 0x7ffd636b426c
Address arr[4] is 0x7ffd636b4270
```

Array vs Pointers
Arrays and pointer are two different things (we can check by applying sizeof). The confusion happens because array name indicates the address of first element and arrays are always passed as pointers (even if we use square bracket). Please see Difference between pointer and array in C? for more details.

What is vector in C++?
Vector in C++ is a class in STL that represents an array. The advantages of vector over normal arrays are,

- We do not need pass size as an extra parameter when we declare a vector i.e, Vectors support dynamic sizes (we do not have to initially specify size of a vector). We can also resize a vector.

- Vectors have many in-built function like, removing an element, etc.
-
- To know more about functionalities provided by vector, please refer vector in C++ for more details.