Arrays

An array holds a fixed number of similar elements that are stored under one name. These elements are stored in contagious memory locations. It is one of the simplest data structures. Most modern programming languages have arrays built-in by default.

one dimensional array

Why use arrays over a bunch of variables?

The reason why we use arrays is that every element can be accessed by its index value. This has several advantages over storing a bunch of variables.

For example: Consider we have to implement a system to store the age of all employees in an office. There is the traditional way with variables.

One can create a variable for each employee in the office. Let's say the office has only 3 employees. Fairly easy right? Just declare 3 variables: emp1_age,emp2_age and emp3_age.

When new recruitments come in, we sit down to create more variables. Maintaining a system like this gets tedious. Imagine one new employee and the whole system code has to be modified.

Accessing each variable would also be a headache. It is stupid to sum 20 variables by hand to calculate the average age of the employees.

An array data structure tries to solve these problems.

One of the properties of arrays is that it holds the same kind of data under one name.

For this example, the array can hold all the ages of the employees under one name, like employees_age. These are all of the integer type.

The second property of arrays is that it stores each element in a continuous block which can be accessed using its index.

Every employee's age can be accessed by iterating through the indices of the array. This can be used to easily access all values serially by looping through them. The function to calculate average becomes much easier to implement as the name of the array is constant and only the index is changing.

Let's see how an array is declared and used.

Declaring a One Dimensional Array

An array has to be declared before it can be used. In C, declaring an array means specifying the following:

Data Type: This is the kind of values that the array will store. This can be characters, integers, floating points or any legal data type.

Name: The variable name used to identify the array and interact with it.

Size: The size of the array, which specifies the maximum number of values that the array will store.

Syntax Used

An array can be declared in C by using the following syntax:

type name[size];

For example, an array of marks of a class of 100 students can be created using:

int marks[100];

There are 2 ways to assign elements to an array:

Assigning values while initialization

The values of the elements can be assigned while declaring the array. If some of the values are not explicitly defined, they are set to 0.

int marks[10] = {5, 10, 20, 30, 40, 60};

Assigning values after initialization

By default, an array is created whenever memory is available at any random location. We do not know what information that random location of memory will contain, as any other program could have used that memory previously.

If array elements are not initialized while creation, then accessing them directly they would result in such garbage values.

Therefore, it is always recommended to empty the elements or assign values to it if a calculation is to be performed on the array.

```
int ages[10];
// accessing array without assigning elements first
for(int i = 0; i < 10; i++)
printf("\n arr[%d] = %d", i, ages[i]);
```

Traversing the array

Each element of the array can be accessed using its index. The indexing in an array generally starts with 0, which means that the first element is at the 0th index. Subsequently, the last element of the array would be at the (n-1)th index. This is known as 0-based indexing.

The indexing of the array may also be different by using any

other base. These are known as n-based indexing.

Accessing all the elements is possible by using a simple for-loop going through all the indices in the array.

```
for(int i = 0; i < arraySize; i++)
printf("\n arr[%d] = %d", i, arr[i]);
```

Example: Values are first being assigned and then displayed from the array.

```
int id[10];
// assigning values using a loop
for (int i = 0; i < 10; i++) {
printf("\nEnter an id: ");
scanf("%d", &id[i]);
}
// displaying the entered ids
for (int i = 0; i < 10; i++) {
printf("\n id[%d] = %d", i, id[i]);
}
```

Maintaining the order of an array while inserting or deleting

requires manipulating the others already present in the array. This is one of the disadvantages, as such operations can be costly on larger arrays.

Inserting an element in the array

At the end

Inserting an element at the end of the array is easy provided the array has enough space for the new element. The index of the last element of the array is found out and the new element is inserted at the index + 1 position.

At any other position

An element can be inserted in between at any position by shifting all elements from that position to the back of the array. The element to be inserted is then inserted at the required position.

```
void insert_position(int arr[]) {
int i = 0, pos, num;
```

```
printf("Enter the number to be inserted  :");
scanf("%d", &num);
printf("Enter position at which the number is to be added  :");
scanf("%d", &pos);
for (i = n-1; i>= pos; i--)
arr[i+1] = arr[i];
arr[pos] = num;
n = n + 1;   /*increase total number of used positions
display_array(arr);
}
```

Deleting an element from the array

At the end

Deleting an element at the end of the array is equally easy provided there is some element to begin with (not an empty array). The index of the last element is found out and this element is deleted.

At any other position

An element can be deleted at any index by deleting the element

at that position and then moving up all the elements from the back of the array to the front to fill up the position of the deleted element.

```
void delete_position(int arr[]) {
int i, pos;
printf("\nEnter the position where the number has to be deleted: ");
scanf("%d", &pos);
for (i = pos; i < n-1; i++)
arr[i] = arr[i+1];
n = n - 1;   //decrease total number of used positions
display_array(arr);
}
```

Multi-Dimensional Arrays

An array may have more than one dimension to represent data. These are known as multidimensional arrays. The elements in these arrays are accessed using multiple indices.

Two Dimensional Array

A two dimensional array can be considered as an array within

an array. It can be visualised as a table, having a row and a column. Each item in the table can be accessed using 2 indices corresponding to the row and column.

Example of 2D array and its indices

Example of a 2D array and its indices

Pluke [CC0] from Wikimedia Commons

A 2D array is declared using 2 parameters:

type name[max_size_x][max_size_y]

The max_size_x and max_size_y are the max values each dimension can store.

Three Dimensional Array

A three dimensional array similarly can be visualised as a cube. Each item can be accessed using 3 indices corresponding to the 3D position.

Example: Every block of a Rubik's Cube can be represented by a three dimensional array of size 3 x 3 x 3.

A 3D array is declared using 2 parameters:

type name[max_size_x][max_size_y][max_size_z];
The max_size_x, max_size_y and max_size_z are the max values each dimension can store.

Memory Allocation in arrays
Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is, the address of the first element is sufficient. The address of other elements can be then calculated using the base address.

For one dimensional array
A simple formula consisting of the size of the element and the lower bound is used.

A[i] = base_address(A) + size_of_element(i - lower_bound)
The lower bound is the smallest index in the array. Similarly, an upper bound is the largest index in the array. In C programming, the lower bound value may be omitted as it is

generally 0.

For two dimensional array

The elements in a two dimensional array can be stored using 2 representations and their addresses can be calculated using the respective formulae.

Column Major Representation

In this form, the elements are stored column by column. m elements of the first column are stored in the first m locations, elements of the second column element are stored in the next m locations, and so on.

Address(A[I][J]) = base_address + width {number_of_rows (J - 1) + (I - 1)}

Row Major Formula

In this form, the elements are stored row by row. n elements of the first row are stored in the first n locations, elements of the second row elements are stored in the next n locations, and so on.

Address(A[I][J]) = base_address + width {number_of_cols (I - 1) + (J - 1)}

Column and Row Major representation of an array

Column and Row Major representation of an array

Cmglee [CC BY-SA 4.0]

Time Complexity of Operations

Access

Any array element could be accessed directly through its index. Hence the access time is constant O(1).

Search

Searching for a given value through the array requires iterating through each element in the array until the element is found. This is assuming that linear search is used (which is the most basic type of search to find any element). This makes the search time O(n).

The other more efficient search algorithm, binary search could be used to search in O(log n) time but it requires the array to be sorted beforehand.

Insertion

Inserting an element in between 2 elements in an array involves shifting all the elements to the right by 1. This means that at most all the elements have to be shifted right (insertion at the beginning of the array), hence the complexity of the insert operation in $O(n)$.

Deletion

Deleting an element in between 2 elements in an array involves shifting all the elements to the left by 1. This means that at most all the elements have to be shifted left (deletion at the beginning of the array), hence the complexity of the delete operation in $O(n)$.

Space Required

An array only takes the space used to store the elements of the data type specified. This means that for storing n elements the space required is $O(n)$.

Advantages of arrays

Arrays have various advantages over other more complex data structures.

Arrays allow for random access of elements. Each element in the array can be interacted with by directly accessing to its index.
Arrays have good cache locality, which means the speed of execution of code may be significantly faster in some cases due to nature how arrays are stored.

Disadvantages of arrays
The size of an array is fixed once declared. This may be insufficient or more than required later on the program. In case the size is inefficient, it may be costly to move all the array elements to a new bigger array.
Insertion and deletion of elements in the array so that it maintains a continuous order may be expensive, as one may have to relocate all the elements.