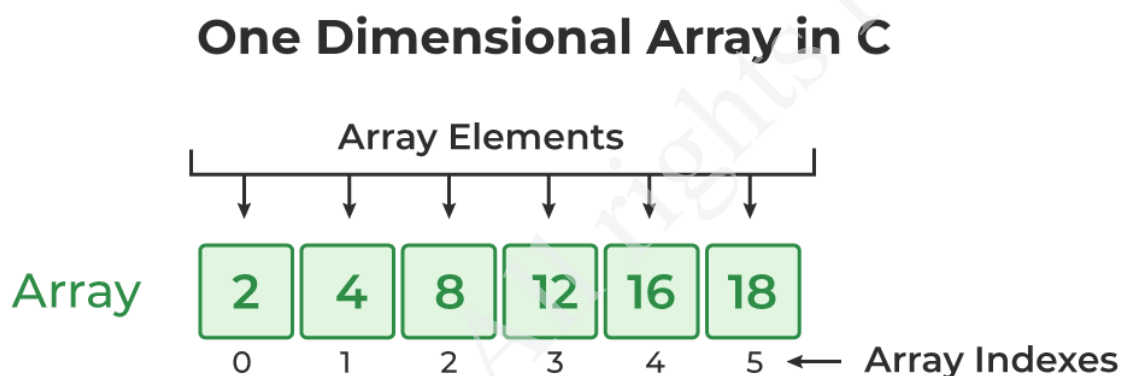


## One Dimensional Arrays in C

In C, an array is a collection of elements of the same type stored in contiguous memory locations. This organization allows efficient access to elements using their index. Arrays can also be of different types depending upon the direction/dimension they can store the elements. It can be 1D, 2D, 3D, and more. We generally use only one-dimensional, two-dimensional, and three-dimensional arrays.

## One-Dimensional Arrays in C

A one-dimensional array can be viewed as a linear sequence of elements. We can only increase or decrease its size in a single direction.



Only a single row exists in the one-dimensional array and every element within the array is accessible by the index. In C, array indexing starts zero-indexing i.e. the first element is at index 0, the second at index 1, and so on up to  $n-1$  for an array of size  $n$ .

## Syntax of One-Dimensional Array in C

The following code snippets show the syntax of how to declare a one-dimensional array and how to initialize it in C.

### 1D Array Declaration Syntax

In declaration, we specify the name and the size of the 1d array.

```
elements_type array_name[array_size];
```

In this step, the compiler reserved the given amount of memory for the array but this step does not define the value of the elements. They may contain some random values. So we initialize the array to give its elements some initial value

## 1D Array Initialization Syntax

In declaration, the compiler reserved the given amount of memory for the array but does not define the value of the element. To assign values, we have to initialize an array.

```
elements_type array_name[array_size] = {value1, value2, ...};
```

This type of The values will be assigned sequentially, means that first element will contain value1, second value2 and so on.

**Note:** The number of elements initialized should be equal to the size of the array. If more elements are given in the initializer list, the compiler will show a warning and they will not be considered.

This initialization only works when performed with declaration

## Array Initialization

```
int arr[5] = {2, 4, 8, 12, 16};
```



```
arr = {2, 4, 8, 12, 16};
```



## 1D Array Element Accessing/Updating Syntax

After the declaration, we can use the index of the element along with the array name to access it.

```
array_name [index];    // accessing the element
```

Then, we can also assign the new value to the element using assignment operator.

```
array_name [index] = new_value;    // updating element
```

Note: Make sure the index lies within the array or else it might lead to segmentation fault.

To access all the elements of the array at once, we can use the loops as shown below.

## Example of One Dimensional Array in C

The following example demonstrate how to create a 1d array in a c program.

```
// C program to illustrate how to create an array,
```

```
// initialize it, update and access elements
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    // declaring and initializing array
```

```
    int arr[5] = { 1, 2, 4, 8, 16 };
```

```
// printing it
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}

printf("\n");

// updating elements
arr[3] = 9721;

// printing again
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}

return 0;
}
```

### Output

```
1 2 4 8 16
```

```
1 2 4 9721 16
```

## Memory Representation of One Dimensional Array in C

Memory for an array is allocated in a contiguous manner i.e. all the elements are stored adjacent to its previous and next element in the memory. It means that, if the address of the first element and the type of

the array is known, the address of any subsequent element can be calculated arithmetically.

The address of **the  $i$ th element** will be calculated as given the base address (**base**) and element size (**size**):

Address of  $i$ th element = **base + (i x size)**

This structured memory layout makes accessing array elements efficient.

**Note:** The name of the array is the pointer to its first element. And also, we don't need to multiply the size of the data type. The compiler already do it for you

### Example: To Demonstrate the 1D Array Memory Layout

// C++ program to illustrate the memory layout of one

// dimensional array

#include <stdio.h>

int main()

{

    // declaring array

    int arr[5] = { 11, 22, 33, 44, 55 };

    // pointer to the first element of the array;

    int\* base = &arr[0];

    // size of each element

    int size = sizeof(arr[0]);

```
// accessing element at index 2

printf("arr[2]: %d\n*(base + 2): %d\n", arr[2],
      *(base + 2));

printf("Address of arr[2]: %p\n", &arr[2]);
printf("Address of base + 2: %p", base + 2);

return 0;
}
```

### Output

```
arr[2]: 33
*(base + 2): 33
Address of arr[2]: 0x7fff1841c3c8
Address of base + 2: 0x7fff1841c3c8
```

As we can see, the `arr[2]` and `(base + 2)` are essentially the same.

## Array of Characters (Strings)

There is one popular use of array that is the 1D array of characters that is used to represent the textual data. It is more commonly known as strings. The strings are essentially an array of character that is terminated by a NULL character (`'\0'`).

### Syntax

```
char name[] = "this is an example string"
```

Here, we don't need to define the size of the array. It will automatically deduce it from the assigned string.

```
// C++ Program to illustrate the use of string
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    // defining array
```

```
    char str[] = "This is GeeksforGeeks";
```

```
    printf("%s", str);
```

```
    return 0;
```

```
}
```

### Output

```
This is GeeksforGeeks
```

## Multidimensional Arrays in C - 2D and 3D Arrays

A **multi-dimensional array in C** can be defined as an array that has more than one dimension. Having more than one dimension means that it can grow in multiple directions. Some popular multidimensional arrays include 2D arrays which grows in two dimensions, and 3D arrays which grows in three dimensions.

### Syntax

The general form of declaring **N-dimensional arrays** is shown below:

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

- **type:** Type of data to be stored in the array.

- **arrName**: Name assigned to the array.
- **size1, size2,..., sizeN**: Size of each dimension.

// Two-dimensional array

```
int two_d[10][20];
```

// Three-dimensional array:

```
int three_d[10][20][30];
```

## Size of Multidimensional Arrays

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of both dimensions.

Consider the array **arr[10][20]**:

- The array **int arr[10][20]** can store total of  $(10 \times 20) = 200$  elements.

To get the size in bytes, we multiply the size of a single element (in bytes) by the total number of elements in the array.

- The size of array **int arr[10][20]** =  $10 * 20 * 4 = 800$  bytes, where the size of int is 4 bytes.

## Types of Multidimensional Arrays

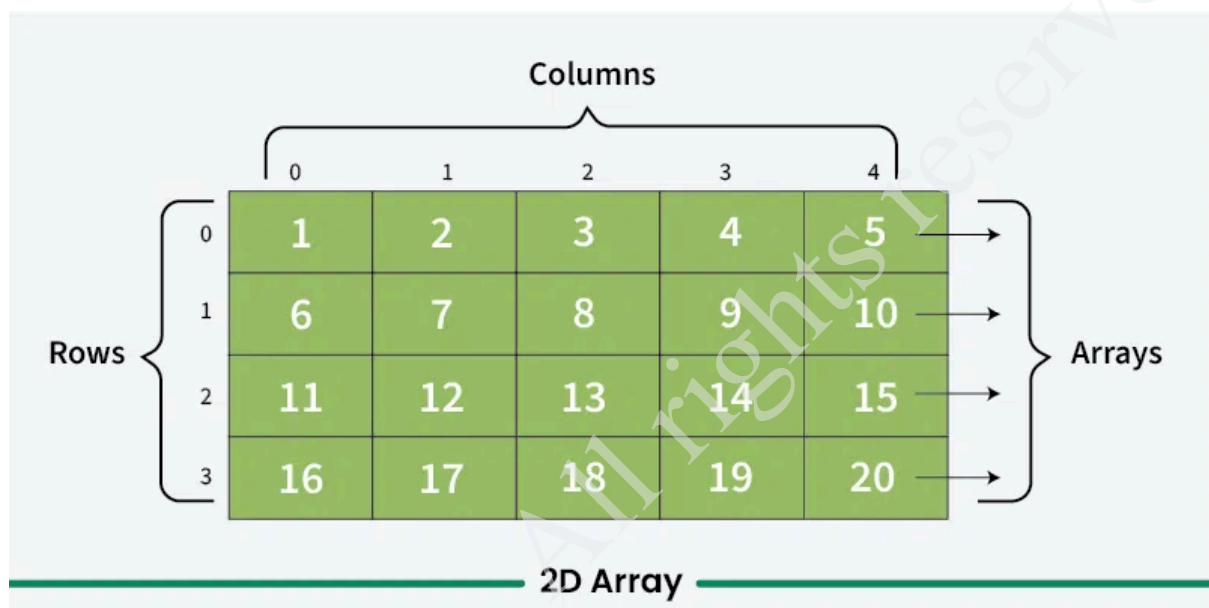
In C, there can be many types of arrays depending on their dimensions but two of them are most commonly used:

1. 2D Array - Two Dimensional
2. 3D Array - Three Dimensional



## 2D Arrays in C

A **two-dimensional array** or **2D array** is the simplest form of the multidimensional array. We can visualize a two-dimensional array as one-dimensional arrays stacked vertically forming a table with 'm' rows and 'n' columns. In C, arrays are 0-indexed, so the row number ranges from 0 to (m-1) and the column number ranges from 0 to (n-1).



### Declaration of 2D Array

A 2D array with **m** rows and **n** columns can be created as:

```
type arr_name[m][n];
```

For example, we can declare a two-dimensional integer array with name '**arr**' with 10 rows and 20 columns as:

```
int arr[10][20];
```

### Initialization of 2D Arrays

We can [initialize a 2D array](#) by using a list of values enclosed inside '{ }' and separated by a comma as shown in the example below:

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

Or

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

The elements will be stored in the array from left to right and top to bottom. So, the first 4 elements from the left will be filled in the first row, the next 4 elements in the second row, and so on. This is clearly shown in the second syntax where each set of inner braces represents one row.

In list initialization, we can skip specifying the size of the row. The compiler will automatically deduce it in this case. So, the below declaration is valid.

```
type arr_name[][n] = {...values...};
```

It is still compulsory to define the number of columns.

**Note:** The number of elements in initializer list should always be either less than or equal to the total number of elements in the array.

## Accessing Elements

An element in two-dimensional array is accessed using row indexes and column indexes inside array subscript [operator \[\]](#).

```
Arr_name[i][j]
```

## 2D Array Traversal

Traversal means accessing all the elements of the array one by one. We will use two loops, outer loop to go over each row from top to bottom and the inner loop is used to access each element in the current row from left to right.

```
for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
        arr[i][j];
    }
}
```

```
}
```

The below example demonstrates the row-by-row traversal of a 2D array.

```
#include <stdio.h>
```

```
int main() {
```

```
    // Create and initialize an array with 3 rows
```

```
    // and 2 columns
```

```
    int arr[3][2] = { { 0, 1 }, { 2, 3 }, { 4, 5 } };
```

```
    // Print each array element's value
```

```
    for (int i = 0; i < 3; i++) {
```

```
        for (int j = 0; j < 2; j++) {
```

```
            printf("arr[%d][%d]: %d  ", i, j, arr[i][j]);
```

```
        }
```

```
        printf("\n");
```

```
    }
```

```
    return 0;
```

```
}
```

### Output

```
arr[0][0]: 0    arr[0][1]: 1
```

```
arr[1][0]: 2    arr[1][1]: 3
```

```
arr[2][0]: 4    arr[2][1]: 5
```

## How 2D Arrays are Stored in the Memory?

As an array, the elements of the 2D array have to be stored contiguously in memory. As the computers have linear memory addresses, the 2-D arrays must be linearized so as to enable their storage. There are two ways to achieve this:

- **Row Major Order:** This technique stores the 2D array row after row. It means that the first row is stored first in the memory, then the second row of the array, then the third row, and so on.
- **Column Major Column:** This technique stores the 2D array column after column. It means that first column is stored first in the memory, then the second column, then the third column, and so on.

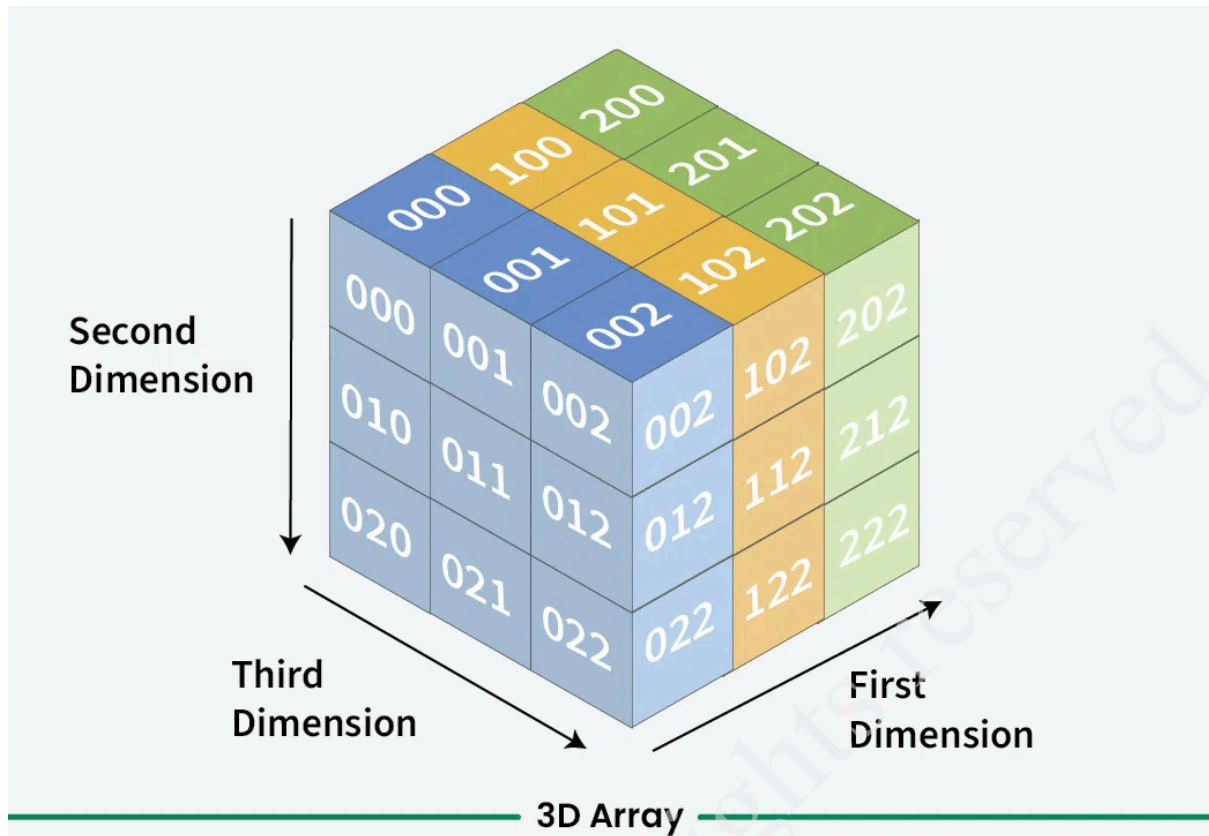
This allows the random access to the 2D array elements as the computer need not keep track of the addresses of all the elements of the array. It only tracks of the Base Address (starting address of the very first element) and [calculates the addresses of other array elements using the base address.](#)

## Passing 2D Arrays to Functions

[Passing 2D arrays to functions](#) need a specialized syntax so that the function knows that the data being passed is 2d array. The function signature that takes 2D array as argument is shown below:

## 3D Array in C

A **Three-Dimensional Array** or **3D array in C** is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



## Declaration of 3D Array in C

We can declare a 3D array with **x** 2D arrays each having **m** rows and **n** columns using the syntax shown below:

```
type arr_name[x][m][n];
```

For example, we can declare 3d array, which is made by 2-2D array and each 2D array have 2 rows and 2 columns:

```
int arr[2][2][2];
```

## Initialization of 3D Array in C

Initialization in a 3D array is the same as that of 2D arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase

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

Or

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

Again, just like the 2D arrays, we can also declare the 3D arrays without specifying the size of the first dimensions if we are using initializer list. The compiler will automatically deduce the size of the first dimension. But **we still need to specify the rest of the dimensions.**

```
arr[][2][2] = {...Values...};
```

## Accessing Elements

To access elements in 3D array, we use three indexes. One for depth, one for row and one for column inside subscript operator [].

```
Arr_name[d][i][j]
```

## 3D Array Traversal

To traverse the entire 3D array, you need to use three nested loops: an outer loop that goes through the depth (or the set of 2D arrays), a middle loop goes through the rows of each 2D array and at last an inner loop goes through each element of the current row.

where, d, i and j are the indexes for depth (representing a specific 2D array.), the row within that 2D array, and the column within that 2D array respectively.

```
for(int d = 0; d < depth; d++){  
  
    for(int i = 0; i < row; i++){  
  
        for(int j = 0; j < col; j++){  
  
            arr[d][i][j];  
  
        }  
  
    }  
  
}
```

Let's take a simple example to demonstrate all the concepts we discussed about 3D arrays:

```
#include <stdio.h>
```

```
int main() {
```

```
    // Create and Initialize the
```

```
    // 3-dimensional array
```

```
    int arr[2][3][2] = { { { 1, 1 }, { 2, 3 },
```

```
                        { 4, 5 } }, { { 6, 7 },
```

```
                        { 8, 9 }, { 10, 11 } } };
```

```
    // Loop through the depth
```

```
    for (int i = 0; i < 2; ++i) {
```

```
        // Loop through the
```

```
        // rows of each depth
```



```
for (int j = 0; j < 3; ++j) {  
  
    // Loop through the  
  
    // columns of each row  
  
    for (int k = 0; k < 2; ++k)  
  
        printf("arr[%i][%i][%i] = %d  ", i, j, k,  
  
            arr[i][j][k]);  
  
    printf("\n");  
  
}  
  
printf("\n\n");  
  
}  
  
return 0;  
  
}
```

## Output

```
arr[0][0][0] = 1   arr[0][0][1] = 1
```

```
arr[0][1][0] = 2   arr[0][1][1] = 3
```

```
arr[0][2][0] = 4   arr[0][2][1] = 5
```

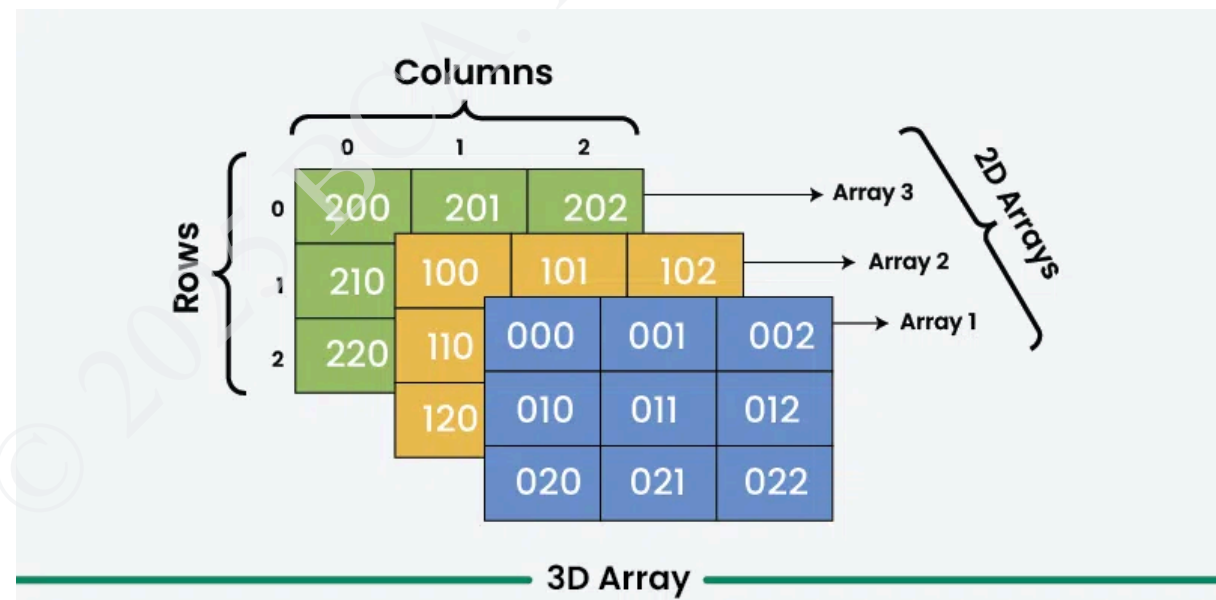
```
arr[1][0][0] = 6   arr[1][0][1] = 7
```

```
arr[1][1][0] = 8   arr[1][1][1] = 9
```

```
arr[1][2][0] = 10  arr[1][2][1] = 11
```

## How 3D Arrays are Stored in the Memory?

Like 2D arrays, the elements of a 3D array should also be stored contiguously in memory.



Since computers have linear memory, the 3D array must also be linearized for storage. We use the same two techniques, the **Row Major Order** and **Column Major Order** but with added dimension. The elements are first stored layer by layer (or 2D array by 2D array). Within each 2D array, the elements follow the corresponding row or column major order.

## Passing 3D Arrays to Functions

[Passing a 3D array to a function](#) in C is similar to passing 2D arrays, but with an additional dimension. When passing a 3D array, you need to pass the sizes of all the dimensions separately because the size information of array is lost while passing.

## C Program for sparse matrix

**Sparse matrix** is a matrix with the majority of its elements equal to zero. However, there is no fixed ratio of zeros to non-zero elements. But it is believed that the number of 0's should be strictly greater than half of the product of rows and columns in a matrix.

In a given matrix, when most of the elements are zero then, we call it as sparse matrix. **Example – 3 x3 matrix**

1 1 0

0 0 2

0 0 0

In this matrix, most of the elements are zero, so it is sparse matrix.

## Problem

Check whether a matrix is a sparse matrix or not.

## Solution

Let us assume ZERO in the matrix is greater than  $(\text{row} * \text{column})/2$ .

Then, the matrix is a sparse matrix otherwise not

## Program

Following is the program to check whether the given matrix is sparse matrix or not –

```
include<stdio.h>
```

```
#include<stdlib.h>
```

```
int main(){
```

```
    int row,col,i,j,a[10][10],count = 0;
```

```
    printf("Enter row
```

```
");
```

```
    scanf("%d",&row);
```

```
    printf("Enter Column
```

```
");
```

```
    scanf("%d",&col);
```

```
    printf("Enter Element of Matrix1
```

```
");
```

```
    for(i = 0; i < row; i++){
```

```
        for(j = 0; j < col; j++){

            scanf("%d",&a[i][j]);

        }

    }

    printf("Elements are:

");

    for(i = 0; i < row; i++){

        for(j = 0; j < col; j++){

            printf("%d\t",a[i][j]);

        }

        printf("

");

    }

    /*checking sparse of matrix*/

    for(i = 0; i < row; i++){

        for(j = 0; j < col; j++){

            if(a[i][j] == 0)

                count++;

        }

    }

}
```

```
    }  
  
}  
  
if(count > ((row * col)/2))  
  
    printf("Matrix is a sparse matrix  
  
");  
  
else  
  
    printf("Matrix is not sparse matrix  
  
");
```

## }Output

When the above program is executed, it produces the following result –

Run 1:

Enter row

3

Enter Column

2

Enter Element of Matrix1

1 0 2 0 2 0

Elements are:

1 0

2 0

2 0

Matrix is not sparse matrix

Run 2:

Enter row

3

Enter Column

2

Enter Element of Matrix1

1 0 0 0 0 0

Elements are:

1 0

0 0

0 0

Matrix is a sparse matrix

## What is a Polynomial?

A **polynomial** is an expression like:

$$5x^3 + 2x^2 + 75x^3 + 2x^2 + 7$$

Here:

- 5 is the coefficient of  $x^3$
  - 2 is the coefficient of  $x^2$
  - 7 is the constant (no  $x$ )
  - The number on top of  $x$  is the **power** or **degree**
- 



## How to Store This in C?

We need a way to **store** this expression in a C program. We can do this in two main ways:

---



### 1. Using an Array (Simple)

We use an array where:

- The **index** is the power of  $x$
- The **value** at that index is the coefficient

**Example:**



Polynomial:

$$5x^3 + 2x^2 + 0x^1 + 7x^0$$

We store it like this:

```
poly[0] = 7; // x^0 term
```

```
poly[1] = 0; // x^1 term
```

```
poly[2] = 2; // x^2 term
```

```
poly[3] = 5; // x^3 term
```

### Code:

```
#include <stdio.h>
```

```
int main() {
```

```
    int poly[4] = {7, 0, 2, 5}; // poly[0] to poly[3]
```

```
    printf("Polynomial: ");
```

```
    for (int i = 3; i >= 0; i--) {
```

```
        if (poly[i] == 0) continue;
```

```
        printf("%dx^%d ", poly[i], i);
```

```
        if (i > 0) printf("+ ");
```

```
    }
```

```
    return 0;
```

```
}
```

---

## ✓ 2. Using a Linked List (For Bigger or Sparse Polynomials)

This method is useful when:

- You don't want to store a lot of 0s
- The polynomial has large powers (like  $x^{100}$ )

We create a **list of terms**, and each term stores:

- Coefficient (like 5)
- Power (like 3)
- A pointer to the next term

### Visual:

$[5x^3] \rightarrow [2x^2] \rightarrow [7x^0] \rightarrow \text{NULL}$

### Code:

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct Node {
    int coeff, power;
```

```
    struct Node* next;
};

struct Node* create(int c, int p) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->coeff = c;
    node->power = p;
    node->next = NULL;
    return node;
}

void print(struct Node* head) {
    while (head) {
        printf("%dx^%d", head->coeff, head->power);
        head = head->next;
        if (head) printf(" + ");
    }
}

int main() {
    struct Node* poly = create(5, 3);
    poly->next = create(2, 2);
    poly->next->next = create(7, 0);

    printf("Polynomial: ");
```

```

print(poly);

return 0;
}

```

---

## Which One Should You Use?

If...	Use...
The polynomial is small	Array
The polynomial has big gaps	Linked List
You want fast access	Array
You want to save memory	Linked List

---

## Stack and Queue C

The stack and queue are popular linear data structures with a wide variety of applications. The stack follows LIFO (Last In First Out) principle where the data is inserted and extracted from the same side. On the other hand, the queue follows FIFO (First In First Out) principle, i.e., data is inserted at one side and extracted from the other side.

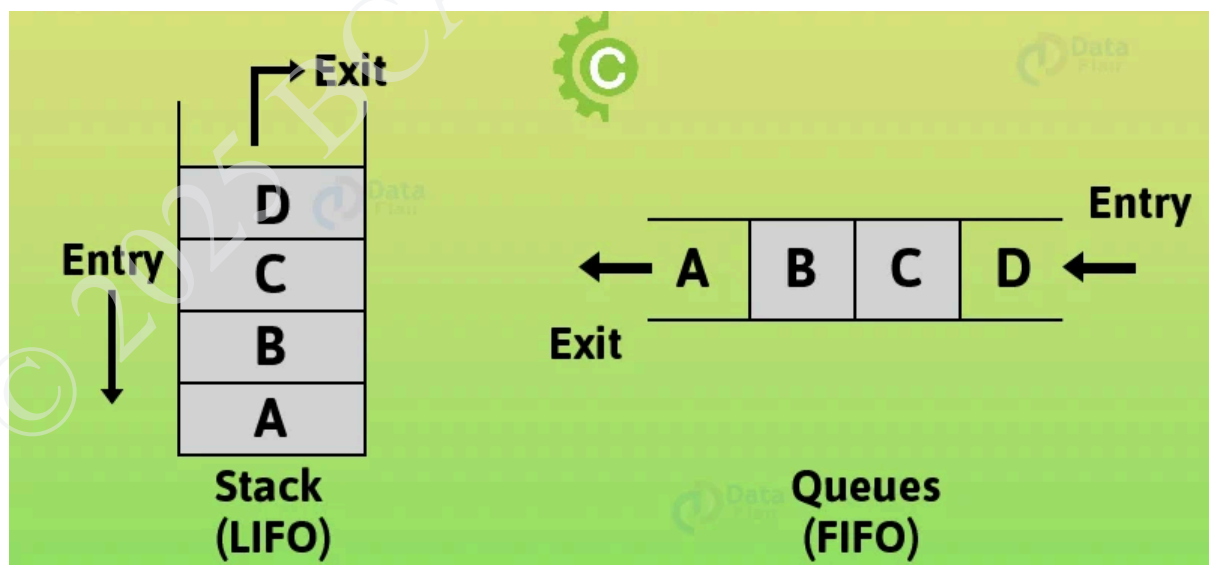
## Stack Practice Problems in C/C++

The following is the list of C/C++ programs based on the level of difficulty:

In this Stacks and Queues in C tutorial, we will discuss:

- Stack in C
- Array implementation of a Stack
- Linked list implementation of a Stack
- Applications of Stack
- Queue in C
- Array implementation of a Queue
- Linked list implementation of a Queue
- Circular Queue
- Applications of Queue

If we want to learn this concept then we have to go deep inside it. That's why we are going to discuss its key topics, which will be helping us to grasp the concept in an efficient way. So let's start –



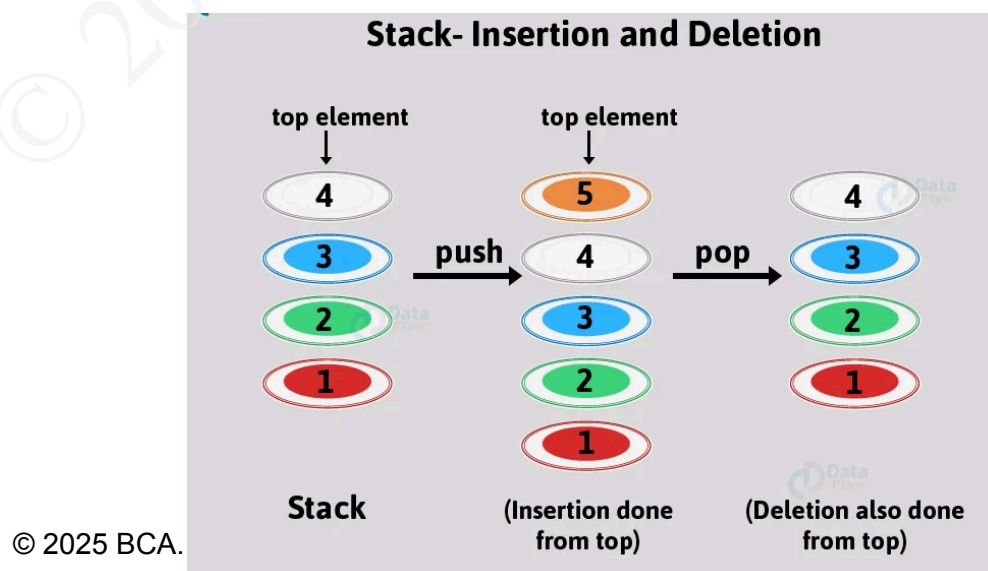
## What is Stack in C?

stack in C is nothing but a linear data structure that follows the LIFO rule (Last In First Out). In a stack, both insertion and deletion take place from just one end, that is, from the top.

In order to better understand it, consider the following scenario: Imagine you have 5 plates that you have to keep on top of each other of distinct colors: Red, Green, Blue, White, and Orange.

You start by placing the red plate on the table. This is the first element of the stack. Then, you place the green plate on top of the red plate. This is the second element of the stack. Similarly, you place the blue plate followed by white and then finally orange. Note that the first plate you inserted into the stack was the red one. Now, you want to remove the red plate. But, before that, you need to remove the rest of the plates that are on top of the red one.

From this discussion, it is pretty obvious that the first plate (first data element) to be inserted is removed at the last. And, the last plate to be inserted is removed at first, that is, it follows the “Last In First Out” rule. Also, we infer that placing and removing the plate is done from the top, that is, insertion and deletion are done from the top



## We can implement a stack in C in 2 ways:

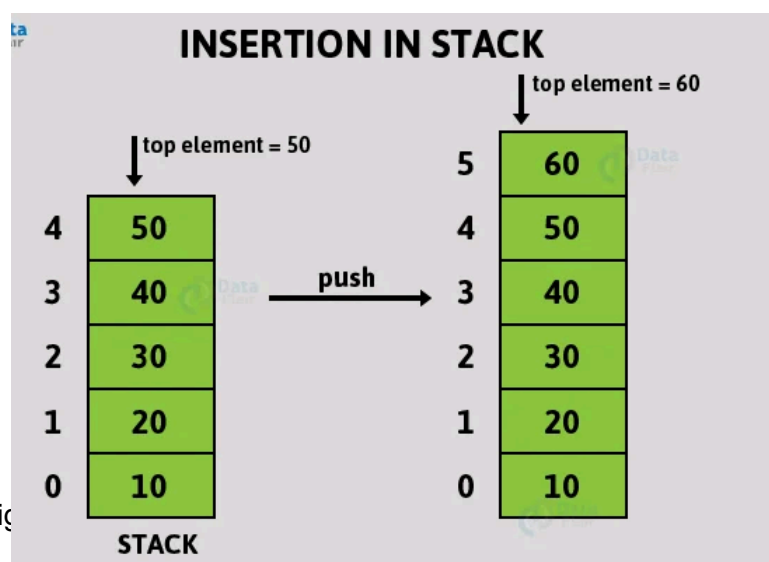
1. **Statically:** Array implementation of stacks allows the static memory allocation of its data elements. It is important to note that in this method, the stack acquires all the features of an array.
2. **Dynamically:** Linked list implementation of stacks follow the dynamic memory allocation of its data elements. It is important to note that in this method, the stack inherits all the characteristics of a [linked list in C](#).

### Array Implementation of Stack in C

As we already discussed, arrays support the static memory allocation of the data elements of the stack. Therefore, it is important to determine the size of the stack prior to the program run

### The C stInsertion

In a stack, the operation of inserting an element into the stack is referred to as pushing an element in the stack. The elements are inserted into the stack from the top and hence would compel the elements to shift. Stack functions basically include:



This is how we can insert elements into a C stack:

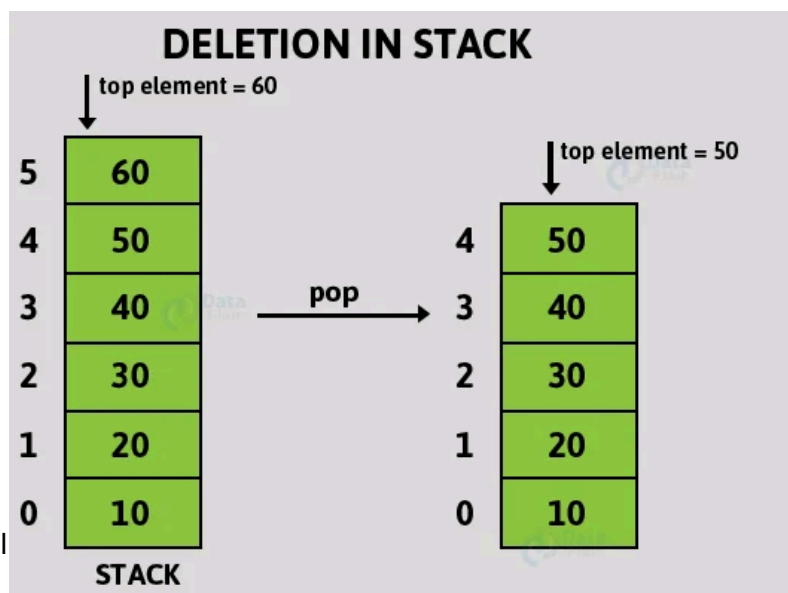
```
#define LIMIT 100
void push()
{
int stack[LIMIT], top, element;
if(top == LIMIT- 1)
{
printf("Stack Overflow\n");
}
else
{
printf("Enter the element to be inserted:");
scanf("%d", &element);
top++;
stack[top]=element;
}
}
```

Enhance your Fundamental Skills with Different types of Operators in C

## Deletion

In a stack, the operation of deleting an element into the stack is referred to as popping an element in the stack. The deletion of a data element from the stack is done from the top.

Here is a diagrammatic representation of how elements are pushed into a stack:





This is how we can delete elements from a stack:

```
#define LIMIT 100
void pop()
{
int stack[LIMIT], top, element;
if(top == -1)
{
printf("Stack underflow\n");
}
else
{
element=stack[top];
printf("The deleted item is %d\n",stack[top]);
top--; // The element below the topmost element is deleted
}
}
```

## Display

The stack data elements are displayed in the stack according to the LIFO rule.

This is how you can display the stack in C:

```
#define LIMIT 100
void display()
{
int stack[LIMIT], top, i;
if(top == -1)
{
printf("Stack underflow\n"); // Stack is empty
}
else if(top > 0)
{
printf("The elements of the stack are:\n");
for(i = top; i >= 0; i--) // top to bottom traversal
{
printf("%d\n",stack[i]);
}
}
}
```

Apart from these 3 main functions, it is necessary to check the overflow and underflow conditions to avoid unfavorable situations.

## Stack Overflow

Here we are talking about the static memory allocation of data elements of a stack. Therefore, if the stack is filled completely, that is, no more elements can be inserted in the stack, then the condition would be called STACK-FULL condition. It is also referred to as stack overflow.

## Stack Underflow

In case we wish to display the data elements of the stack or perform the deletion operation, but no elements have been inserted into the stack yet, this condition is called STACK-EMPTY. It is also referred to as stack underflow.

*Before discussing the example, let's revise the concept of [Variables in C](#)*

Here is a program in C that illustrates the array implementation of stacks:

```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 100 // Specifying the maximum limit of the stack

/* Global declaration of variables */

int stack[LIMIT]; // Array implementation of stack
int top; // To insert and delete the data elements in the stack
int i; // To traverse the loop to while displaying the stack
int choice; // To choose either of the 3 stack operations

void push(); // Function used to insert the element into the stack
void pop(); // Function used to delete the element from the stack
void display(); // Function used to display all the elements in the stack according to LIFO rule

int main()
{
    printf("Welcome to DataFlair tutorials!\n\n");

    printf("ARRAY IMPLEMENTATION USING STACKS\n\n");
    top = -1; // Initializing top to -1 indicates that it is empty
    do
    {
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n\n");
        printf("Enter your choice:");
```

```
scanf("%d",&choice);

switch(choice)
{
case 1:
push();
break;
case 2:
pop();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("Sorry, invalid choice!\n");
break;
}
} while(choice!=4);
return 0;
}

void push()
{
int element;
if(top == LIMIT- 1)
{
printf("Stack underflow\n");
}
else
{
printf("Enter the element to be inserted:");
scanf("%d", &element);
top++;
stack[top]=element;
}
}

void pop()
{
int element;
if(top == -1)
```

```

{
printf("Stack underflow\n");
}
else
{
element=stack[top];
printf("The deleted item is %d\n",stack[top]);
top--; // The element below the topmost element is deleted
}
}

void display()
{
if(top == -1)
{
printf("Stack underflow\n"); // Stack is empty
}
else if(top > 0)
{
printf("The elements of the stack are:\n");
for(i = top; i >= 0; i--) // top to bottom traversal
{
printf("%d\n",stack[i]);
}
}
}
}

```

### Output-

```

dataflair@asus-System-Product-Name: ~/Desktop
dataflair@asus-System-Product-Name:~/Desktop$ gcc astack.c -o astack
dataflair@asus-System-Product-Name:~/Desktop$ ./astack
Welcome to DataFlair tutorials!

ARRAY IMPLEMENTATION USING STACKS

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted:10

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted:20

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted:30

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted:40

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:

```

# Linked List Implementation of Stack in C

As we already discussed, linked lists support the dynamic memory allocation of the data elements of the stack. Therefore, the size of the stack is allocated during the program run and needn't be specified beforehand.

The 3 basic operations of Insertion, Deletion, and Display follow a similar trend as we saw in the array implementation of stacks.

[Functions in C](#) – *An Important Concept for beginners*

The stack functions basically include:

## 1.2.1 Insertion

This is how we can insert elements into the stack in C:

```
void push ()
{
    int data;
    struct node *pointer = (struct node*)malloc(sizeof(struct node));
    if(pointer == NULL)
    {
        printf("Stack overflow");
    }
    else
    {
        printf("Enter the element to be inserted: ");
        scanf("%d",&data);
        if(temp == NULL)
        {
            pointer->data = data;
            pointer->next = NULL;
            temp = pointer;
        }
        else
        {
            pointer->data = data;
            pointer->next = temp;
            temp = pointer;
        }
    }
}
```

```
}
```

## Deletion

This is how void pop()

```
{
int item;

struct node *pointer;

if (temp == NULL)
{
printf("Stack Underflow\n");
}

else
{
item = temp -> data;
pointer = temp;
temp = temp -> next;
free(pointer);
printf("The deleted item is %d\n",item);
}
}
```

we can delete elements from a stack in C:

## Display

This is how we display the elements of a stack:

```
void display()
```

```

{
int i;

struct node *pointer;

pointer = temp;

if(pointer == NULL)

{

printf("Stack underflow\n");

}

else

{

printf("The elements of the stack are:\n");

while(pointer!= NULL)

{

printf("%d\n",pointer -> data);

pointer = pointer -> next;

}

}

}

```

Here is a code in C that illustrates the linked list implementation of arrays:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
void push(); // Function used to insert the element into the stack
```

```
void pop(); // Function used to delete the element from the stack
```

void display(); // Function used to display all the elements in the stack according to LIFO rule

```
struct node
```

```
{
```

```
int data;
```

```
struct node *next;
```

```
};
```

```
struct node *temp;
```

```
int main()
```

```
{
```

```
printf("Welcome to DataFlair tutorials!\n\n");
```

```
int choice;
```

```
printf("LINKED LIST IMPLEMENTATION USING STACKS\n\n");
```

```
do
```

```
{
```

```
printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n\n");
```

```
printf("Enter your choice:");
```

```
scanf("%d",&choice);
```



```
switch(choice)
{
case 1:
push();
break;
case 2:
pop();
break;
case 3:
display();
break;
case 4:
exit(o);
break;
default:
printf("Sorry, invalid choice!\n");
break;
}
} while(choice!=4);
return o;
}
```

```
void push ()
```

```
{
```

```
int data;

struct node *pointer = (struct node*)malloc(sizeof(struct node));

if(pointer == NULL)
{
    printf("Stack overflow");
}
else
{
    printf("Enter the element to be inserted: ");
    scanf("%d",&data);
    if(temp == NULL)
    {
        pointer -> data = data;
        pointer -> next = NULL;
        temp = pointer;
    }
    else
    {
        pointer -> data = data;
        pointer -> next = temp;
        temp = pointer;
    }
}
```

```
void pop()
{
    int item;

    struct node *pointer;

    if (temp == NULL)
    {
        printf("Stack Underflow\n");
    }
    else
    {
        item = temp -> data;
        pointer = temp;
        temp = temp -> next;
        free(pointer);
        printf("The deleted item is %d\n",item);
    }
}

void display()
{
    int i;

    struct node *pointer;

    pointer = temp;

    if(pointer == NULL)
```

```

{
printf("Stack underflow\n");
}
else
{
printf("The elements of the stack are:\n");
while(pointer!= NULL)
{
printf("%d\n",pointer -> data);
pointer = pointer -> next;
}
}
}

```

Here is a code in C that illustrates the linked list implementation of arrays:

```

#include <stdio.h>

#include <stdlib.h>

void push(); // Function used to insert the element into the stack
void pop(); // Function used to delete the element from the stack
void display(); // Function used to display all the elements in the stack according to
LIFO rule

struct node
{

```

```
int data;

struct node *next;

};

struct node *temp;


int main()

{

printf("Welcome to DataFlair tutorials!\n\n");


int choice;

printf ("LINKED LIST IMPLEMENTATION USING STACKS\n\n");

do

{

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n\n");

printf("Enter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:

push();

break;
```

```
case 2:

pop();

break;

case 3:

display();

break;

case 4:

exit(o);

break;

default:

printf("Sorry, invalid choice!\n");

break;

}

} while(choice!=4);

return o;

}

void push ()

{

int data;

struct node *pointer = (struct node*)malloc(sizeof(struct node));

if(pointer == NULL)

{

printf("Stack overflow");
```

```
}  
  
else  
  
{  
  
printf("Enter the element to be inserted: ");  
  
scanf("%d",&data);  
  
if(temp == NULL)  
  
{  
  
pointer -> data = data;  
  
pointer -> next = NULL;  
  
temp = pointer;  
  
}  
  
else  
  
{  
  
pointer -> data = data;  
  
pointer -> next = temp;  
  
temp = pointer;  
  
}  
  
}  
  
}  
  
void pop()  
  
{  
  
int item;  
  
struct node *pointer;
```

```
if (temp == NULL)

{

printf("Stack Underflow\n");

}

else

{

item = temp -> data;

pointer = temp;

temp = temp -> next;

free(pointer);

printf("The deleted item is %d\n",item);

}

}

void display()

{

int i;

struct node *pointer;

pointer = temp;

if(pointer == NULL)

{

printf("Stack underflow\n");

}

else

{
```



```
printf("The elements of the stack are:\n");
```

```
while(pointer!= NULL)
```

```
{
```

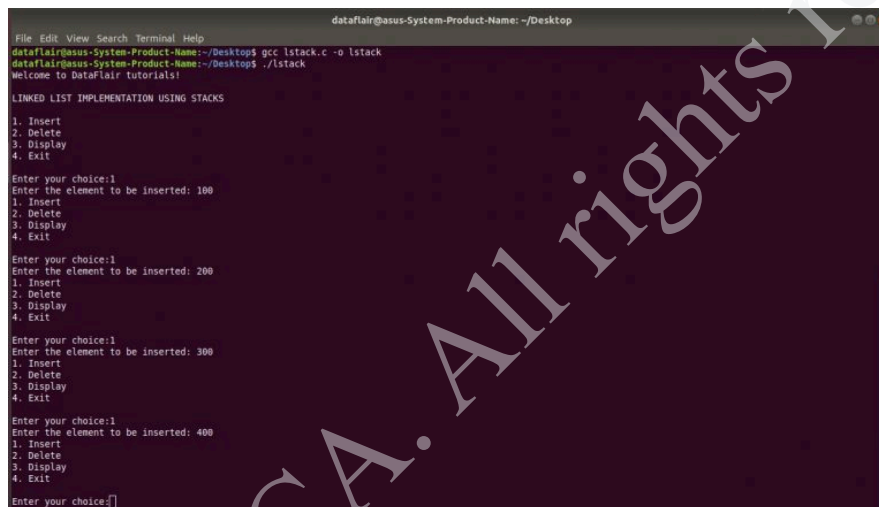
```
printf("%d\n",pointer -> data);
```

```
pointer = pointer -> next;
```

```
}
```

```
}
```

```
}
```



```
dataflair@asus-System-Product-Name: ~/Desktop
dataflair@asus-System-Product-Name:~/Desktop$ gcc lstack.c -o lstack
dataflair@asus-System-Product-Name:~/Desktop$ ./lstack
Welcome to DataFlair tutorials!

LINKED LIST IMPLEMENTATION USING STACKS

1. Insert
2. Delete
3. Display
4. Exit

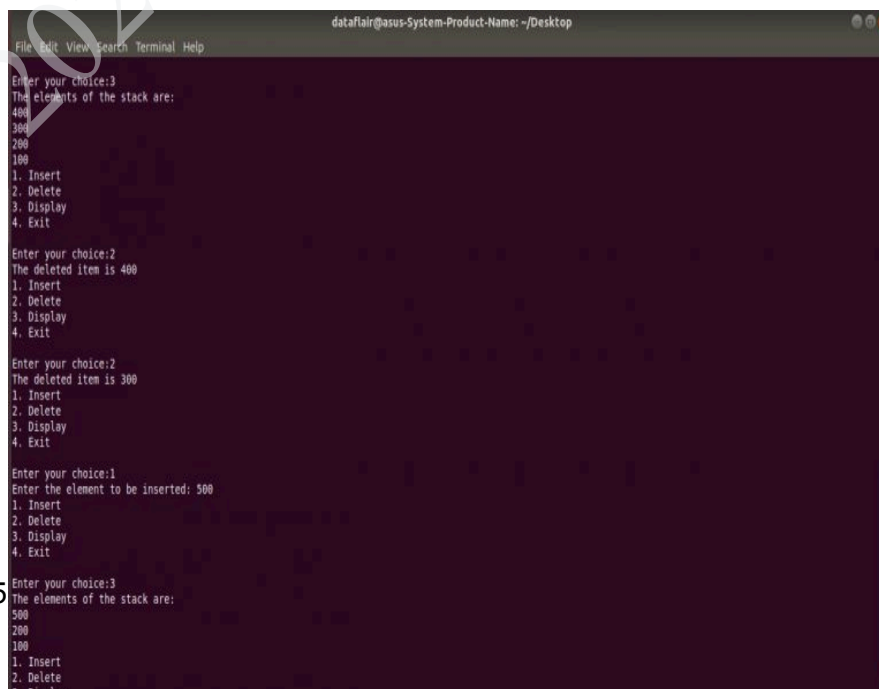
Enter your choice:1
Enter the element to be inserted: 100
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted: 200
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted: 300
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted: 400
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:[]
```



```
dataflair@asus-System-Product-Name: ~/Desktop
Enter your choice:3
The elements of the stack are:
400
300
200
100
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:2
The deleted item is 400
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:2
The deleted item is 300
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:1
Enter the element to be inserted: 500
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice:3
The elements of the stack are:
500
200
100
1. Insert
2. Delete
3. Display
4. Exit
```

# Application of Stack in C

The principle [LIFO](#) followed by stacks gives birth to the various applications of stacks. Some of the most popular applications of stacks are:

- Number reversal: A stack helps you reverse a number or a word entered as a sequence of digits or characters respectively.
- Undo operation: Implementation of a stack helps you perform the “undo” operation in text editors or word processors. Here, all the changes done in the text editor are stored in a stack.
- Infix to postfix conversion: Using stacks, you can perform the conversion of an infix expression to a postfix expression.
- Backtracking: Stacks are finding applications in puzzle or maze problem-solving.
- Depth-first search (DFS): Stacks allow you to perform a searching algorithm called the depth-first search.

## What is a Queue in C?

In contrast to a stack, a *queue in C is nothing but a linear data structure that follows the FIFO rule (First In First Out)*. Insertion is done from the back (the rear end) and deletion is done from the front.

In order to better understand the concept of queues in C, we can say that it follows the rule of “First Come First Serve”. Let us consider a simple scenario to help you get a clear picture of queues. Suppose you want to purchase a movie ticket. For that, you need to stand in a queue and wait for your turn, that is, you have to stand at the rear end of the queue. You can’t simply stand in the middle of the queue or occupy the front position.

From the above discussion, it is pretty obvious that insertion in a queue takes place from the back and deletion is taking place from the front as the first person to enter the queue would be the first to get his job done and leave.

We can implement a queue in 2 ways:

1. **Statically:** Array implementation of queues allows the static memory allocation of its data elements. It is important to note that in this method, the queue acquires all the features of an array.
2. **Dynamically:** Linked list implementation of queues follow the dynamic memory allocation of its data elements. It is important to note that in this method, the queue inherits all the characteristics of a linked list.

**Key takeaway:** Both stacks and queues in C can be static or dynamic according to the way they are implemented.

## Array Implementation of Queue in C

As we already discussed, arrays support the static memory allocation of the data elements of the queue. Therefore, it is important to determine the size of the queue prior to the program run.

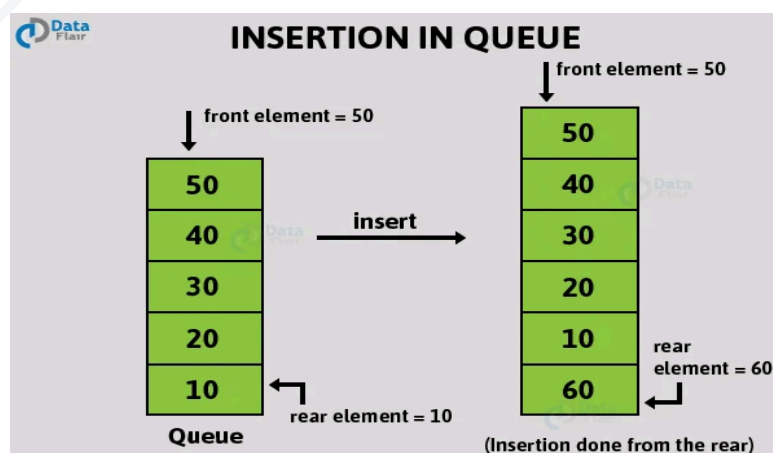
It's the right time to uncover the secrets of Multi-dimensional Arrays in C

The queue functions basically include:

### Insertion

Insertion of elements into the queue takes place from the rear end. Inserting an element into the queue is also called enqueue.

Here is a diagrammatic representation of how elements are inserted into a queue:



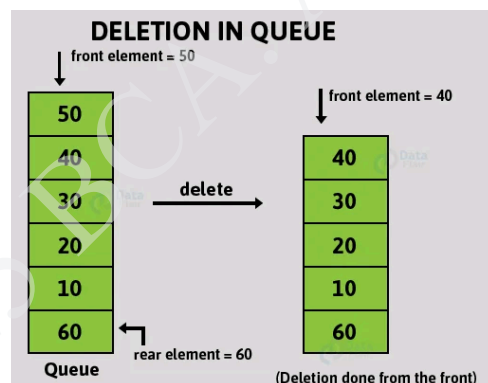
This is how can insert elements into the queue in C:

```
void insert()
{
    int element;
    if (rear == LIMIT - 1)
        printf("Queue Overflow\n");
    else
    {
        if (front == - 1)
            front = 0;
        printf("Enter the element to be inserted in the queue: ");
        scanf("%d", &element);
        rear++;
        queue[rear] = element;
    }
}
```

## Deletion

In a queue, the removal of data elements is done from the front. Removing the element from the queue is also called dequeue

Here is a diagrammatic representation of how elements are deleted from a queue in C



This is how we can delete elements from the queue:

```
void delet()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
    }
    else
    {

```

```
printf("The deleted element in the queue is: %d\n", queue[front]);
front++;
}
}
```

Key takeaway: It is important to note that we used the function name “delet” instead of “delete” because “delete” is a keyword.

## Display

The stack data elements are displayed in the queue according to the FIFO rule.

This is how we display all the elements in the queue in C:

```
void display()
{
    int i;
    if (front == - 1)
    {
        printf("Queue underflow\n");
    }
    else
    {
        printf("The elements of the queue are:\n");
        for (i = front; i <= rear; i++)
            printf("%d\n", queue[i]);
    }
}
```

Similarly, in queues, apart from these 3 main functions, it is necessary to check the overflow and underflow conditions to avoid unfavorable situations.

Here is a code in C that illustrates the array implementation of queues in C:

```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 100 // Specifying the maximum limit of the queue

/* Global declaration of variables */

int queue[LIMIT]; // Array implementation of queue
int front, rear; // To insert and delete the data elements in the queue respectively
```

```

int i; // To traverse the loop to while displaying the stack
int choice; // To choose either of the 3 stack operations

void insert(); // Function used to insert the element into the queue
void delet(); // Function used to delete the element from the queue
void display(); // Function used to display all the elements in the queue according to
FIFO rule

int main()
{

printf("Welcome to DataFlair tutorials!\n\n");

printf ("ARRAY IMPLEMENTATION OF QUEUES\n\n");
front = rear = -1; // Initializing front and rear to -1 indicates that it is empty
do
{

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n\n");
printf("Enter your choice:");
scanf("%d",&choice);

switch(choice)
{
case 1:
insert();
break;
case 2:
delet();
break;
case 3:
display();
break;
case 4:
exit(o);
break;
default:
printf("Sorry, invalid choice!\n");
break;
}
} while(choice!=4);
return o;
}

```

```
void insert()
{
    int element;
    if (rear == LIMIT - 1)
        printf("Queue Overflow\n");
    else
    {
        if (front == - 1)
            front = 0;
        printf("Enter the element to be inserted in the queue: ");
        scanf("%d", &element);
        rear++;
        queue[rear] = element;
    }
}

void delet()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
    }
    else
    {
        printf("The deleted element in the queue is: %d\n", queue[front]);
        front++;
    }
}

void display()
{
    int i;
    if (front == - 1)
    {
        printf("Queue underflow\n");
    }
    else
    {
        printf("The elements of the queue are:\n");
        for (i = front; i <= rear; i++)
            printf("%d\n", queue[i]);
    }
}
```

# Linked List Implementation of Queue in C

As we already discussed, linked lists support the dynamic memory allocation of the data elements of the queue. Therefore, the size of the queue is allocated during the program run and needn't be specified beforehand.

The 3 basic operations of Insertion, Deletion, and Display follow a similar trend as we saw in the array implementation of queues.

It is important to note that the condition of queue overflow does not exist in the linked list implementation of queues and the size of the stack is not pre-determined. But, the queue underflow condition still holds true.

## Insertion

This is how we can insert elements into the queue:

```
void insert()
{
    struct node *temp;

    temp = (struct node*)malloc(sizeof(struct node));
    printf("Enter the element to be inserted in the queue: ");
    scanf("%d", &temp->data);
    temp->link = NULL;
    if (rear == NULL)
    {
        front = rear = temp;
    }
    else
    {
        rear->link = temp;
        rear = temp;
    }
}
```

## Deletion

This is how the deletion is done in a queue:

```
void delet()
```



```

{
struct node *temp;
temp = front;
if (front == NULL)
{
printf("Queue underflow\n");
front = rear = NULL;
}
else
{
printf("The deleted element from the queue is: %d\n", front->data);
front = front->link;
free(temp);
}
}

```

## Display

This is how we display the elements in the queue

```

void display()
{
struct node *temp;
temp = front;
int cnt = 0;
if (front == NULL)
{
printf("Queue underflow\n");
}
else
{
printf("The elements of the stack are:\n");
while (temp)
{
printf("%d\n", temp->data);
temp = temp->link;
cnt++;
}
}
}

```

## Infix, Postfix and Prefix Expressions/Notations

Mathematical formulas often involve complex expressions that require a clear understanding of the order of operations. To represent these expressions, we use different notations, each with its own advantages and disadvantages. In this article, we will explore three common expression notations: **infix**, **prefix**, and **postfix**.

## Infix Expressions

**Infix expressions** are mathematical expressions where the **operator is placed between its operands**. This is the most common mathematical notation used by humans. For example, the expression " $2 + 3$ " is an infix expression, where the operator "+" is placed between the operands "2" and "3".

**Infix notation** is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account, and parentheses can be used to override the default order of operations.

### Common way of writing Infix expressions:

- Infix notation is the notation that we are most familiar with. For example, the expression " $2 + 3$ " is written in infix notation.
- In infix notation, operators are placed between the operands they operate on. For example, in the expression " $2 + 3$ ", the addition operator "+" is placed between the operands "2" and "3".
- Parentheses are used in infix notation to specify the order in which operations should be performed. For example, in the expression " $(2 + 3) * 4$ ", the parentheses indicate that the addition operation should be performed before the multiplication operation.

## Operator precedence rules:

**Infix expressions** follow operator precedence rules, which determine the order in which operators are evaluated. For example, multiplication and division have higher precedence than addition and subtraction. This means that in the expression " $2 + 3 * 4$ ", the multiplication operation will be performed before the addition operation.

Here's the table summarizing the operator precedence rules for common mathematical operators:

Operator	Precedence
Parentheses ()	Highest
Exponents ^	High
Multiplication *	Medium
Division /	Medium

Addition +	Low
Subtraction -	Low

## Evaluating Infix Expressions

Evaluating infix expressions requires additional processing to handle the order of operations and parentheses. First convert the **infix expression** to **postfix notation**. This can be done using a [stack](#) or a recursive algorithm. Then evaluate the postfix expression.

## Advantages of Infix Expressions

- More natural and easier to read and understand for humans.
- Widely used and supported by most programming languages and calculators.

## Disadvantages Infix Expressions

- Requires parentheses to specify the order of operations.
- Can be difficult to parse and evaluate efficiently.

## Prefix Expressions (Polish Notation)

**Prefix expressions** are also known as **Polish notation**, are a mathematical notation where the operator precedes its operands. This differs from the more common **infix notation**, where the operator is placed between its operands.

In prefix notation, the operator is written first, followed by its operands. For example, the infix expression "a + b" would be written as "+ a b" in prefix notation.

## Evaluation of Prefix Expression

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions. As long as we can guarantee that a valid prefix or postfix expression is used, it can be evaluated with correctness.

We can convert [infix to postfix](#) and can convert [infix to prefix](#).

In this article, we will discuss how to evaluate an expression written in prefix notation. The method is similar to evaluating a postfix expression. Please read [Evaluation of Postfix Expression](#) to know how to evaluate postfix expressions

## Infix, Prefix and Postfix Expressions

When you write an arithmetic expression such as  $B * C$ , the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator \* appears between them in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.

Consider another infix example,  $A + B * C$ . The operators + and \* still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the \* take B and C? The expression seems ambiguous.