What is Stack Structure in C?

A stack is a linear data structure which follows LIFO (last in first out) or FILO (first in last out) approach to perform a series of basic operation, ie. Push, Pop, atTop, Traverse, Quit, etc. A stack can be implemented using an array and linked list.

Stack Operations in C

There are two basic operations performed in stack:

1) Push: Adds an element in the stack.

2) Pop: Removes an element from the stack.

- Note: The elements always popped and pushed in the opposite order.

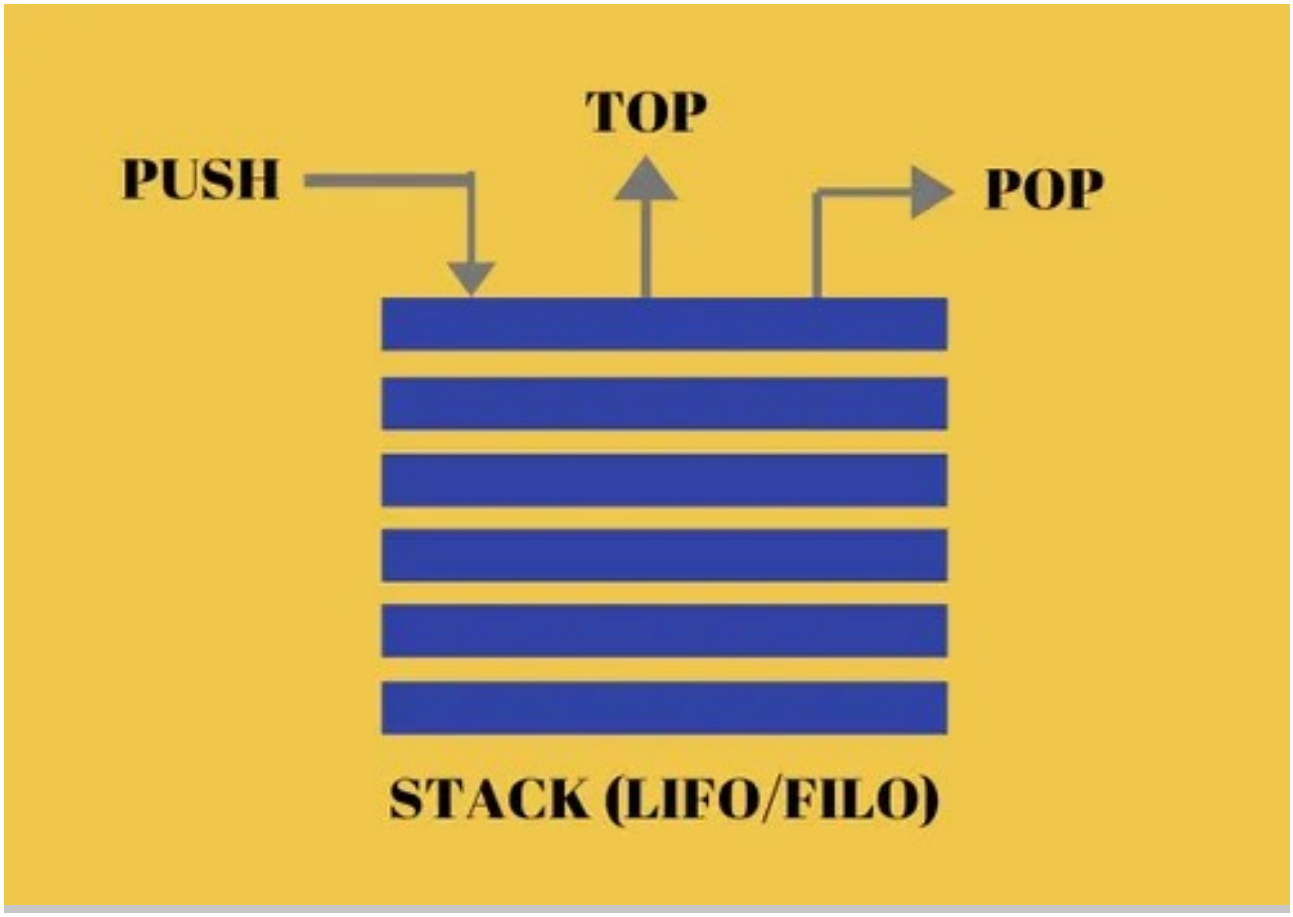
Some Other Important Stack Operations are

- **isEmpty:** checks whether the stack is empty or not
- **atTop:** It returns the top element present in the stack
- **Traverse:** This operation process all the elements present in stack exactly once.
- **Quit:** Quit the stack

Stack Overflow and Underflow

- **Stack Overflow:** It is a condition that happens when we try to push more elements into the already saturated stack
- **Stack Underflow:** It is a condition that happens when we try to pop an element from an empty stack

How does Stack Works?



A stack is a limited access data structure because push (addition) and pop (deletion) occur only at one end of the structure known as the top of the stack. Push adds an item to the top of the stack, pop removes the item from the top.

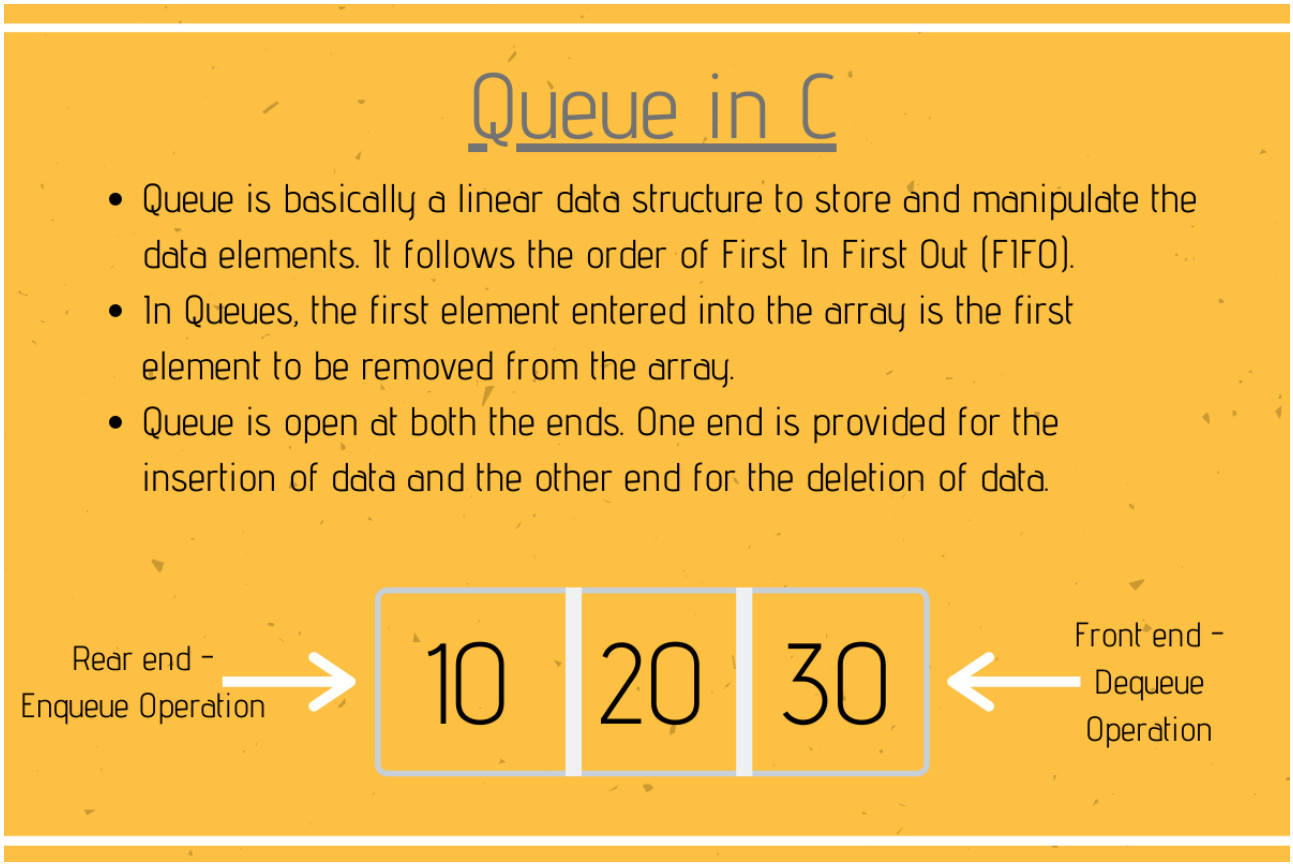
```
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}
```

```
int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}

int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is
empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.
\n");
   }
}

int main() {
   // push items on to the stack
   push(3);
   push(5);
```

```
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack:
%d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" ,
isfull()?"true":"false");
   printf("Stack empty: %s\n" ,
isempty()?"true":"false");

   return 0;
}
```

Output:

```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```



Operations Associated with a Queue in C

A queue being an **Abstract Data Structure** provides the following operations for manipulation on the data elements:

- `isEmpty()`: To check if the queue is empty
- `isFull()`: To check whether the queue is full or not
- `dequeue()`: Removes the element from the frontal side of the queue
- `enqueue()`: It inserts elements to the end of the queue
- `Front`: Pointer element responsible for fetching the first element from the queue
- `Rear`: Pointer element responsible for fetching the last element from the queue

Working of Queue Data Structure

Queue follows the First-In-First-Out pattern. The first element is the first to be pulled out from the list of elements.

- `Front` and `Rear` pointers keep the record of the first and last element in the queue.
- At first, we need to initialize the queue by setting `Front = -1` and `Rear = -1`
- In order to insert the element (**enqueue**), we need to check whether the queue is already full i.e. **check the condition for Overflow**. If the queue is not full, we'll have to increment the value of the Rear index by 1 and place the element at the position of the Rear pointer variable. When we get to insert the first element in the queue, we need to set the value of Front to 0.
- In order to remove the element (**dequeue**) from the queue, we need to check whether the queue is already empty i.e. **check the condition for Underflow**. If the queue is not empty, we'll have to remove and return the element at the position of the Front pointer, and then increment the Front index value by 1. When we get **to remove the last element from the queue**, we will have to **set the values of the Front and Rear index to -1.**

```
#include <stdio.h>
# define SIZE 100
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
main()
{
   int ch;
   while (1)
   {
      printf("1.Enqueue Operation\n");
```

```
        printf("2.Dequeue Operation\n");
        printf("3.Display the Queue\n");
        printf("4.Exit\n");
        printf("Enter your choice of operations : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
            enqueue();
            break;
            case 2:
            dequeue();
            break;
            case 3:
            show();
            break;
            case 4:
            exit(0);
            default:
            printf("Incorrect choice \n");
        }
    }
}

void enqueue()
{
    int insert_item;
    if (Rear == SIZE - 1)
       printf("Overflow \n");
    else
    {
        if (Front == - 1)

        Front = 0;
        printf("Element to be inserted in the Queue\n : ");
        scanf("%d", &insert_item);
        Rear = Rear + 1;
        inp_arr[Rear] = insert_item;
```

```
    }
}

void dequeue()
{
    if (Front == - 1 || Front > Rear)
    {
        printf("Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from the Queue: %d\n",
inp_arr[Front]);
        Front = Front + 1;
    }
}

void show()
{

    if (Front == - 1)
        printf("Empty Queue \n");
    else
    {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
            printf("%d ", inp_arr[i]);
        printf("\n");
    }
}
```