Implement two stacks in an array

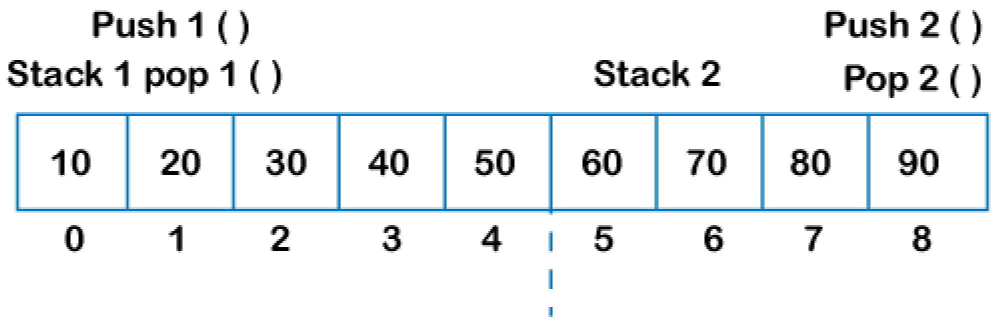
Here, we will create two stacks, and we will implement these two stacks using only one array, i.e., both the stacks would be using the same array for storing elements.

There are two approaches to implement two stacks using one array:

First Approach

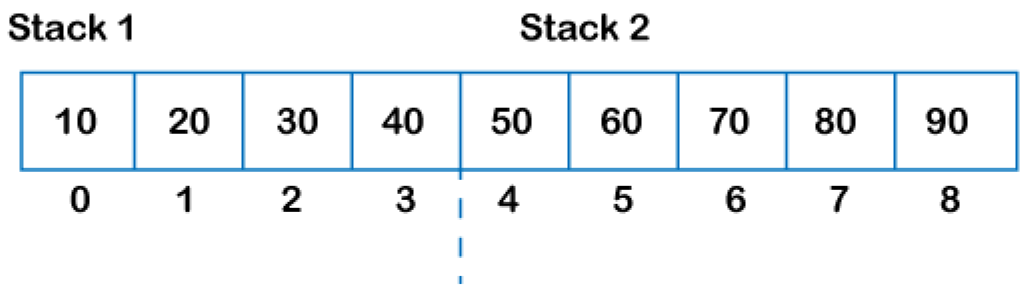
First, we will divide the array into two sub-arrays. The array will be divided into two equal parts. First, the sub-array would be considered stack1 and another sub array would be considered stack2.

For example, if we have an array of n equal to 8 elements. The array would be divided into two equal parts, i.e., 4 size each shown as below:



The first subarray would be stack 1 named as st1, and the second subarray would be stack 2 named as st2. On st1, we would perform push1() and pop1() operations, while in st2, we would perform push2() and pop2() operations. The stack1 would be from 0 to n/2, and stack2 would be from n/2 to n-1.

If the size of the array is odd. For example, the size of an array is 9 then the left subarray would be of 4 size, and the right subarray would be of 5 size shown as below:



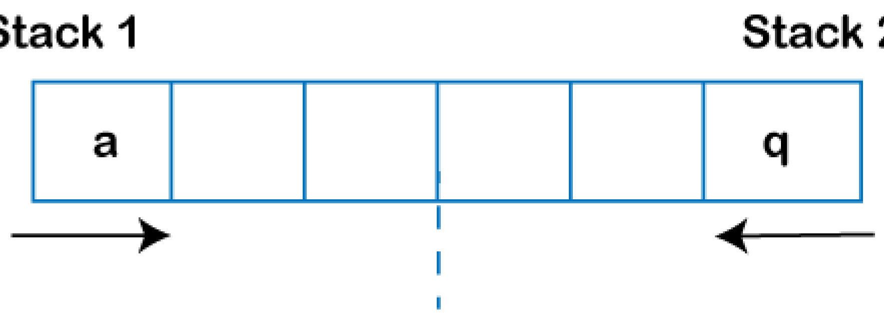
Disadvantage of using this approach

Stack overflow condition occurs even if there is a space in the array. In the above example, if we are performing push1() operation on the stack1. Once the element is inserted at the 3rd index and if we try to insert more elements, then it leads to the overflow error even there is a space left in the array.

Second Approach

In this approach, we are having a single array named as 'a'. In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction, shown as below:

If we push 'a' into stack1 and 'q' into stack2 shown as below:



Therefore, we can say that this approach overcomes the problem of the first approach. In this case, the stack overflow condition occurs only when **top1 + 1 = top2**. This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

Implementation in C

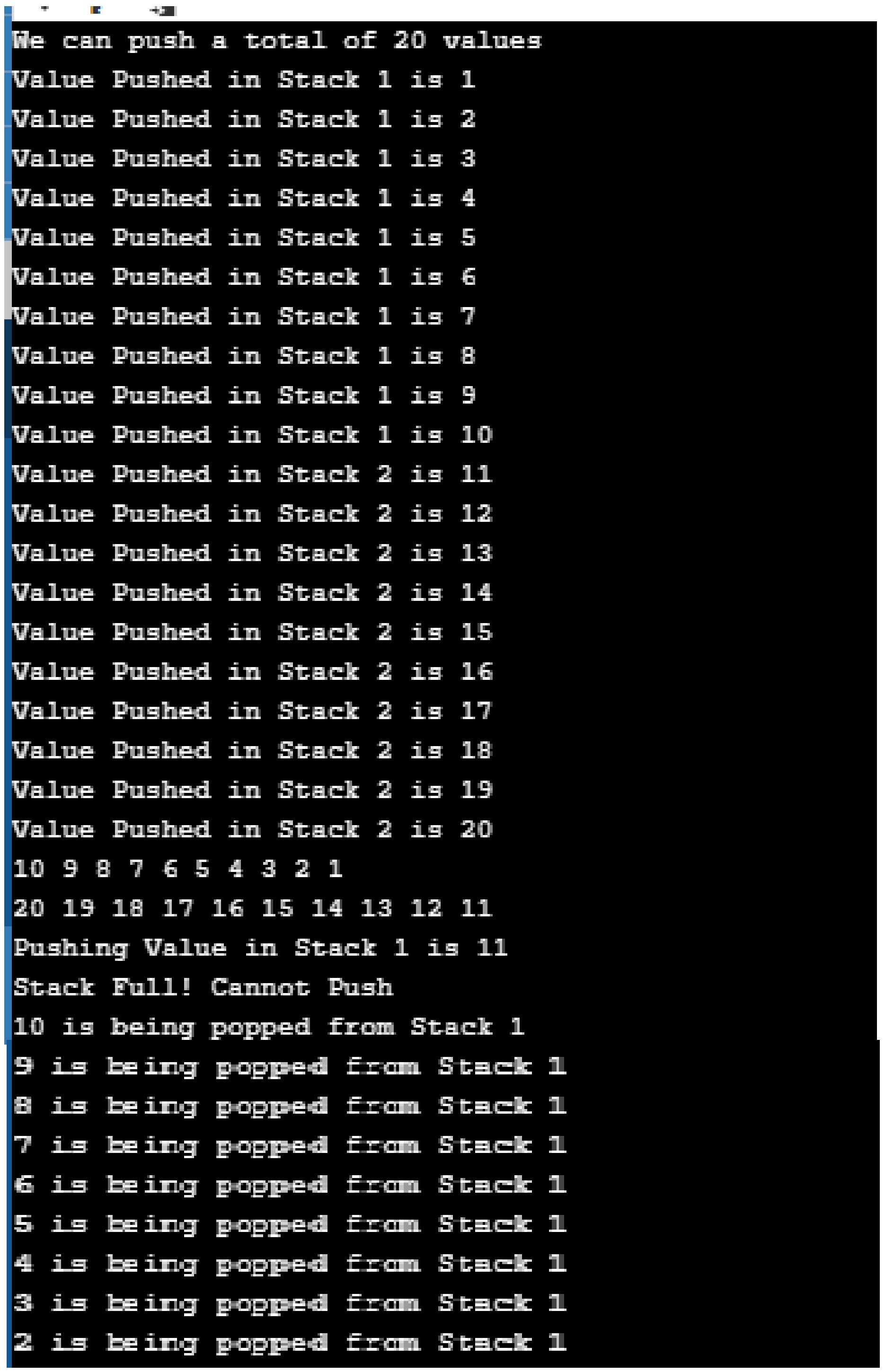
// C Program to Implement two Stacks using a Single Array & Check for Overflow & Underflow

```
#include <stdio.h>
#define SIZE 20
 int array[SIZE];  // declaration of array type variable.
int top1 = -1;
int top2 = SIZE;

//Function to push data into stack1
void push1 (int data)
{
// checking the overflow condition
  if (top1 < top2 - 1)
  {
      top1++;
    array[top1] = data;
  }
  else
  {
    printf ("Stack is full");
  }
}
// Function to push data into stack2
void push2 (int data)
{
// checking overflow condition
if (top1 < top2 - 1)
  {
    top2--;
    array[top2] = data;
  }
  else
  {
    printf ("Stack is full..\n");
  }
}

//Function to pop data from the Stack1
void pop1 ()
{
// Checking the underflow condition
 if (top1 >= 0)
  {
    int popped_element = array[top1];
    top1--;

    printf ("%d is being popped from Stack 1\n", popped_element);
  }
  else
  {
    printf ("Stack is Empty \n");
  }
}
// Function to remove the element from the Stack2
void pop2 ()
{
// Checking underflow condition
if (top2 < SIZE)
  {
      int popped_element = array[top2];
    top2--;

    printf ("%d is being popped from Stack 1\n", popped_element);
  }
  else
  {
    printf ("Stack is Empty!\n");
  }
}

//Functions to Print the values of Stack1
void display_stack1 ()
{
  int i;
  for (i = top1; i >= 0; --i)
  {
    printf ("%d ", array[i]);
  }
  printf ("\n");
}
// Function to print the values of Stack2
void display_stack2 ()
{
  int i;
  for (i = top2; i < SIZE; ++i)
  {
    printf ("%d ", array[i]);
  }
  printf ("\n");
}

int main()
{
  int ar[SIZE];
  int i;
  int num_of_ele;

  printf ("We can push a total of 20 values\n");

  //Number of elements pushed in stack 1 is 10
```

```
//Number of elements pushed in stack 2 is 10

        // loop to insert the elements into Stack1
        for (i = 1; i <= 10; ++i)
          {
            push1(i);
            printf ("Value Pushed in Stack 1 is %d\n", i);
          }
        // loop to insert the elements into Stack2.
        for (i = 11; i <= 20; ++i)
          {
            push2(i);
            printf ("Value Pushed in Stack 2 is %d\n", i);
          }

          //Print Both Stacks
          display_stack1 ();
         display_stack2 ();

          //Pushing on Stack Full
          printf ("Pushing Value in Stack 1 is %d\n", 11);
          push1 (11);

          //Popping All Elements from Stack 1
          num_of_ele = top1 + 1;
          while (num_of_ele)
          {
            pop1 ();
            --num_of_ele;
          }

          // Trying to Pop the element From the Empty Stack
          pop1 ();

```

```
    return 0;
}
```

Output



```
We can push a total of 20 values
Value Pushed in Stack 1 is 1
Value Pushed in Stack 1 is 2
Value Pushed in Stack 1 is 3
Value Pushed in Stack 1 is 4
Value Pushed in Stack 1 is 5
Value Pushed in Stack 1 is 6
Value Pushed in Stack 1 is 7
Value Pushed in Stack 1 is 8
Value Pushed in Stack 1 is 9
Value Pushed in Stack 1 is 10
Value Pushed in Stack 2 is 11
Value Pushed in Stack 2 is 12
Value Pushed in Stack 2 is 13
Value Pushed in Stack 2 is 14
Value Pushed in Stack 2 is 15
Value Pushed in Stack 2 is 16
Value Pushed in Stack 2 is 17
Value Pushed in Stack 2 is 18
Value Pushed in Stack 2 is 19
Value Pushed in Stack 2 is 20
10 9 8 7 6 5 4 3 2 1
20 19 18 17 16 15 14 13 12 11
Pushing Value in Stack 1 is 11
Stack Full! Cannot Push
10 is being popped from Stack 1
9 is being popped from Stack 1
8 is being popped from Stack 1
7 is being popped from Stack 1
6 is being popped from Stack 1
5 is being popped from Stack 1
4 is being popped from Stack 1
3 is being popped from Stack 1
2 is being popped from Stack 1
```

Implement Multi Stack (K stacks) using only one Data Structure

A dynamic multi-stack is a remarkable data structure that possesses the capacity to store elements in numerous stacks, with an ever-changing quantity of stacks. It can be a daunting task to implement K stacks utilizing only one data structure. In this instructional guide, we shall investigate two distinct techniques to execute dynamic multi-stack (K stacks) using C++. The initial technique employs an array to stock the elements, along with two additional arrays to monitor the topmost and following indices of the stacks. The secondary technique employs a vector of nodes to stock the elements, along with a vector to keep track of the head of every stack.

This article will center upon how we may execute dynamic multi-stack employing one data structure in C++.

Approaches

- **Approach 1** − Using an array to store the elements of the data structure and two auxiliary arrays to store the top element of each stack and the next index of the free slot in the array.
- **Approach 2** − Using a doubly linked list to store the elements of the data structure and a vector to store the head node of each stack.

Syntax

The given syntax is the declaration of a class named KStacks in C++. The class has the following member functions or methods −

- A constructor method KStacks which takes two parameters k and n.
- A method named push which takes two parameters item and sn and is used to insert an element into stack sn.
- A method named pop which takes a single parameter sn and is used to remove an element from stack sn.
- A method named is_empty which takes a single parameter sn and returns a boolean value indicating whether stack sn is empty or not.
- A method named is_full which returns a boolean value indicating whether the entire data structure is full or not.

```
class KStacks {
  public:
  KStacks(int k, int n); // Constructor
  void push(int item, int sn); // Insert an element into stack 'sn'
  int pop(int sn); // Remove an element from stack 'sn'
  bool is_empty(int sn); // Check if stack 'sn' is empty
  bool is_full(); // Check if the entire data structure is full
};
```

Algorithm

The following is an algorithm to implement a K-stack dynamic multipack using a single data structure −

- **Step 1** − Start by creating a data structure that contains an array of size n to store the elements, along with two auxiliary arrays of size k. One array will store the information about the topmost element of each stack, while the other array will keep track of the next available index in the main array.
- **Step 2** − Next, we call the parent array and its corresponding array using the values -1 and 0.
- **Step 3** − Using the cart() function, we can add an object to a specific stack. This function requires two inputs: the item to be added and the group number. Before adding an item, the push() function checks if the data structure is full by comparing the next available index value with n. If there is still space, the item is added to the next available index and the value of the next available index is updated.
- **Step 4** − The pop() function is used to remove items from a particular stack, with the group number as its parameter. The pop() function checks if the stack is empty by comparing the parent array value with -1. If the stack is not empty, the pop() function removes the topmost element from the stack and updates the parent array value to point to the new topmost element.
- **Step 5** − To check if a specific stack is empty, we use the is_empty() function with the group number as its parameter. This function checks if the parent array value is equal to -1.
- **Step 6** − To check if all the stacks are full, we use the is_full() function, which checks if the next available index value is equal to n.

Approach 1

We shall adopt an approach that involves utilizing an array to reserve the elements, as well as two additional arrays to monitor the topmost and subsequent indices of the stacks. Although it is a straightforward and effective solution, it necessitates the predefinition of a predetermined number of stacks.

Below is the program code for the same.

Example

The code represents the implementation of the K Stacks data structure, which is a dynamic interpretation of the stack data structure, allowing for multiple stacks to be housed within a single array.

The KStacks class comprises three member variables −

- **arr** − An array serving as storage for the elements of all stacks.
- **top** − An array serving as storage for the top of each stack.
- **next** − An array serving as storage for the next available position in the array.
- **push** − To insert an element into a specified stack.
- **pop** − To remove an element from a specified stack.
- **is_empty** − To verify if a specified stack is empty.
- **is_full** − To verify if the array is completely occupied.

In the main function, an instance of the KStacks class is generated, with the number of stacks and size of the array as input parameters. Then, elements are pushed into three distinct stacks. Lastly, the top elements from each stack are removed and displayed.

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

class KStacks {
  private:
  int *arr;
  int *top;
  int *next;
  int n, k;
  public:
  KStacks(int k, int n) {
    this->k = k;
    this->n = n;
    arr = new int[n];
    top = new int[k];
    next = new int[n];
  for (int i = 0; i < k; i++)
    top[i] = -1;

  for (int i = 0; i < n - 1; i++)
    next[i] = i + 1;
  next[n - 1] = -1;
}

void push(int item, int sn) {
  if (is_full()) {
    cout << "Stack Overflow\n";
    return;
  }

  int i = next[sn];
  next[sn] = top[sn];
  top[sn] = i;
  arr[i] = item;
}

int pop(int sn) {
  if (is_empty(sn)) {
    cout << "Stack Underflow\n";
    return INT_MAX;
  }

  int i = top[sn];
  top[sn] = next[i];
```

```
    next[i] = i;
    return arr[i];
    }

    bool is_empty(int sn) {
      return top[sn] == -1;
    }

    bool is_full() {
      return next[0] == -1;
    }
};

int main() {
    KStacks ks(3, 10);
    ks.push(15, 2);
    ks.push(45, 2);

    ks.push(17, 1);
    ks.push(49, 1);
    ks.push(39, 1);

    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);

    cout << "Popped element from stack 2: " << ks.pop(2) << endl;
    cout << "Popped element from stack 1: " << ks.pop(1) << endl;
    cout << "Popped element from stack 0: " << ks.pop(0) << endl;
  return 0;

}
```

Output

```
Stack Overflow
Stack Overflow
Popped element from stack 2: Stack Underflow
2147483647
Popped element from stack 1: 39
           Popped element from stack 0: 11
```