TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:

Lab No: 1

Title:

Submitted By

Name: Anish Karki

Roll No: PAS080BCT007

Submitted To: HPB Sir

Date of submission: **12/06/2025**

PRACTICAL-1
INSERTION OF AN ARRAY

THEORY

Inserting elements into an array in C++ involves carefully moving existing elements to make room for the new one. In this program, we handle insertion at any valid position provided by the user. The elements starting from the chosen index are shifted one place towards the end to create space for the new element.

Insertion Steps

Select the position to insert.
Shift all elements from the end to that position to the right.
Insert the new element at the target index and update the array size.

Boundary Conditions:

Insertion at the beginning requires shifting all elements.
Insertion at the end is direct if space exists.
Be careful of array overflow.

Time Complexity:

Insert at end: O (1)
Insertion, not from end: O(n)

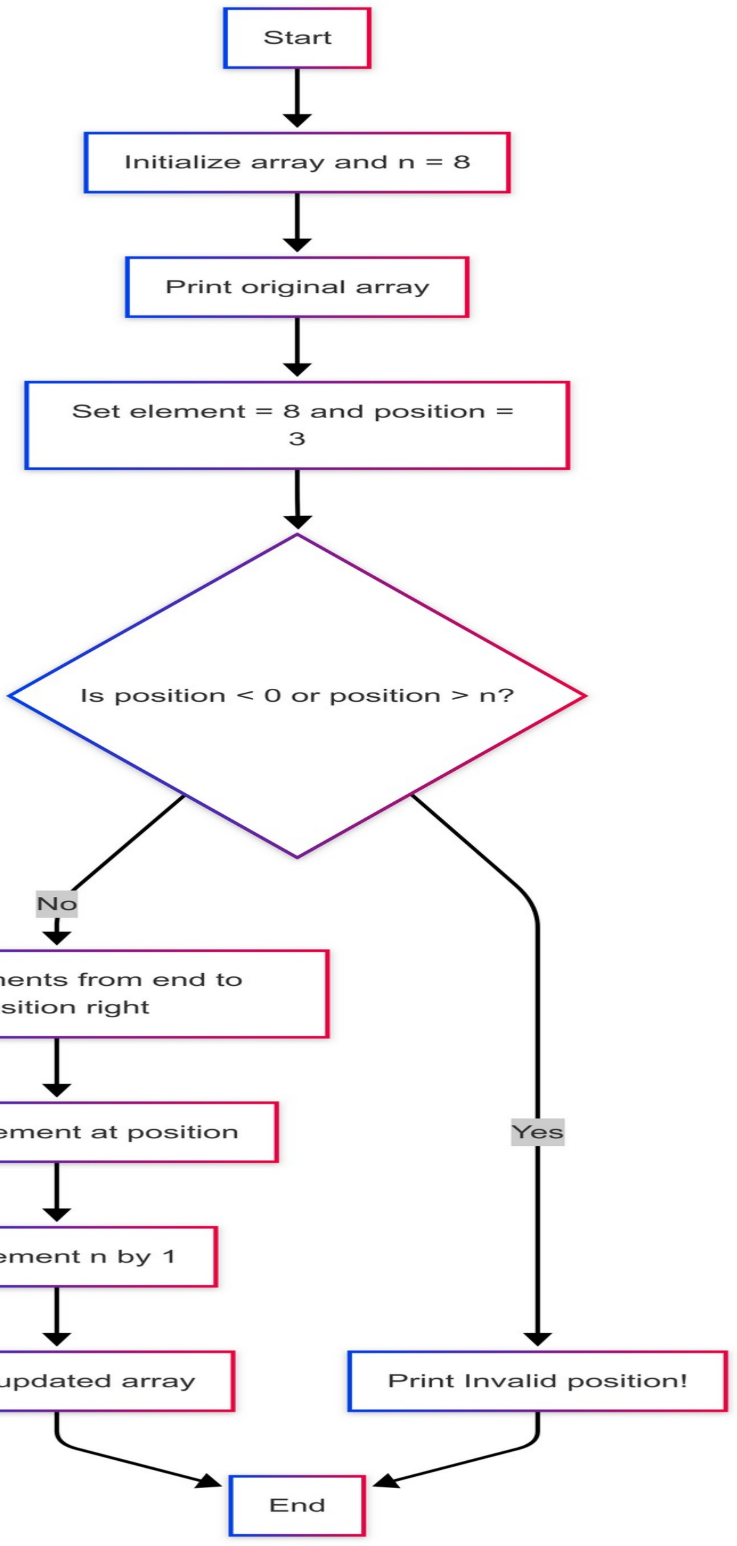
Since the array size is fixed, inserting beyond its capacity leads to overflow. Vectors from STL can handle dynamic resizing and avoid this issue.

ALGORITHM

Steps:

```
1) Start
2) Initialize Array and size
3) Print Original Array
        •Print Original Array
        •Call printArray(arr, n)
4) Set Element and Position
        •element = 8
        •position = 3
5) Call
        insertElement(arr, n, element, position)
        •Check if Position is Valid
                •If position < 0 || position > n:
                        •Print "Invalid position!"
                        •Return
        •Shift Elements to the Right
                •Loop: for (int i = n; i > position; i--)
                        •arr[i] = arr[i - 1]
        •Insert Element
                •arr[position] = element
        •Increment Array Size
                •n++
6) Print Array After Insertion
        •Call printArray(arr, n)
7)End
```

FLOWCHART


Start
Initialize array and n = 8
Print original array
Set element = 8 and position = 3
Is position < 0 or position > n?
No
Shift elements from end to position right
Insert element at position
Increment n by 1
Print updated array
Yes
Print Invalid position!
End

CODE:

```
#include <iostream>
 using namespace std;
 void insertElement(int arr[], int &n, int element, int position) {
            if (position < 0 || position > n) {
                cout << "Invalid position!" << endl;
                return;
        }

     for (int i = n; i > position; i--) {
                arr[i] = arr[i - 1]; }
     arr[position] = element; n++;
        }

 void printArray(int arr[], int n) {
        for (int i = 0; i < n; i++) {
            cout << arr[i] << " ";
            }
             cout << endl;
}


int main() {
   int arr[10] = {3, 5, 7, 9, 11, 13, 15, 17};
   int n = 8;

   cout << "Original array: ";
   printArray(arr, n);

   int element = 8, position = 3;

   insertElement(arr, n, element, position);

 cout << "Array after insertion: ";


  printArray(arr, n);
  return 0;

}
```

PRACTICAL-2
DELETION OF AN ARRAY

THEORY

Deleting an element from an array in C++ involves shifting elements to fill the gap created by the removed element. In this program, deletion is done at any valid index given by the user. Elements after the deleted position are shifted one place to the left, and the size of the array is reduced by one.

Deletion Steps

Check if the given index is valid.
Shift all elements after the index one place to the left.
Decrease the array size by one..

Boundary Conditions:

Insertion at the beginning requires shifting all elements.
Insertion at the end is direct if space exists.
Be careful of array overflow.

Time Complexity:

Deletion at any position: **O(n)**, as shifting may be required.

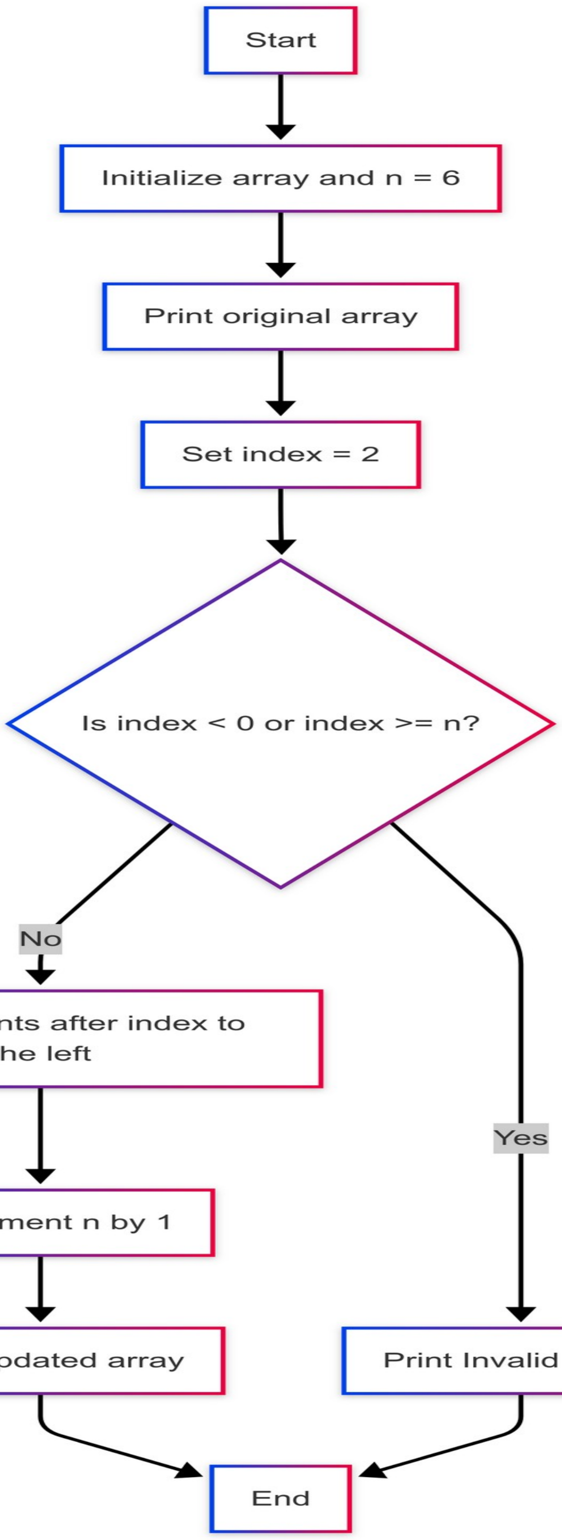
Since the array size is fixed, we can't shrink or expand it dynamically. Dynamic structures like **vectors** (C++ STL) are more flexible for such operations..

ALGORITHM

Steps:

```
1)Start
2) Initialize Array and size
3) Print Original Array
    •Print Original Array
    •Call printArray(arr, n)
4) Set index for deletion
    •index = 2
5) Call
     deleteElement(arr, n, index)
    •Check if Position is Valid
        •If position < 0 || position > n:
            •Print "Invalid position!"
            •Return
    •Shift Elements to the Left
        •Loop: for (int i = index; i < n; i++)
            •arr[i] = arr[i +1]
    •Decrement Array Size
        •n--
6) Print Array After Deletion
    •Call printArray(arr, n)
7)End
```

FLOWCHART


Start
Initialize array and n = 6
Print original array
Set index = 2
Is index < 0 or index >= n?
No
Shift elements after index to the left
Decrement n by 1
Print updated array
Yes
Print Invalid index!
End

CODE:

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

void deleteElement(int arr[], int& n, int index) {
        if (index < 0 || index >= n) {
            cout << "Invalid index!" << endl;
            return;
           }
        for (int i = index; i < n - 1; i++) {
               arr[i] = arr[i + 1];
         }
         n--;
}

void printArray(int arr[], int n) {
         for (int i = 0; i < n; i++) {
         cout << arr[i] << " ";
         }
         cout << endl;
 }

int main() {
              int arr[] = {7, 9, 11, 13, 15, 17};
              int n = 6;

              cout << "Original array: ";
              printArray(arr, n);

              int index = 2;

              deleteElement(arr, n, index);

              cout << "Array after deletion: ";
              printArray(arr, n);


             return 0;
}
```

CONCLUSION

In this lab, we implemented both insertion and deletion operations on arrays using C++. Insertion involved shifting elements to the right to create space, while deletion required shifting elements to the left to fill the gap. Although arrays provide fast access, their fixed size makes dynamic operations less flexible. Using dynamic structures like vectors can simplify such tasks in real-world applications.