Arrays in C++

By: Aaryan Vinod Kumar

An array is a data structure that lets you store multiple values of the same type using a single name. Instead of creating individual variables, you can use one array to store all the values in an organized way.

Why Arrays though?

- **Organized Storage**: Arrays help store multiple values of the same type under one name, making code cleaner and easier to manage.
- **Fast Access**: You can access any element instantly using its index, without having to search through the entire data.
- **Efficient Memory Use**: Since arrays store data in consecutive memory blocks, they use memory efficiently and allow for quick calculations of element positions.
- **Foundation for Other Structures**: Arrays form the basis for more complex data structures like matrices, strings, stacks, queues, and more.

Syntax:

```
datatype array_name[size];
```

Example:

```
int marks[5] = {90, 85, 78, 92, 88};
```

- This declares an array called marks that holds 5 integers.
- Each value can be accessed using an index: marks[0] gives 90, marks[1] gives 85, etc.

Note: What does "stored together" mean?

Arrays store their elements next to each other in memory, which is called contiguous memory.
Think of it like 5 boxes kept side by side on a shelf, each holding one value.
The computer knows exactly where each box is, and can quickly access any of them using the index.

Behind the Scenes: How Arrays Work

When you declare:

```
int arr[5] = {99, -192, 47, 420, 69};
```

This is how the data is arranged in memory:

Index	Value
0	99
1	-192
2	47
3	420
4	69

Note: What is Contiguous Memory?

Contiguous memory means all elements are stored **one after the other** in memory. This lets the computer directly calculate the address of any element using its index, without searching through each item.

For example, if the first element is stored at address `100`, and each integer takes 4 bytes, then:

- `arr[1]` will be at address `104`
- `arr[2]` will be at address `108`
  and so on.

Basic Operations on Arrays

Traversal

Definition: Going through each element of the array one by one.

Code:

```
int marks[5] = {90, 85, 78, 92, 88};


for (int i = 0; i < 5; i++) {

    cout << marks[i] << " ";

}
```

Explanation:

- The loop runs from index 0 to 4.
- Each element is printed using `cout`.

Insertion

Definition: Adding a new element at a specific position in the array.

Code:

```
int arr[10] = {1, 2, 3, 4, 5};
int n = 5;  // Current number of elements
int pos = 2;  // Index where new element is to be inserted
int val = 99;

for (int i = n; i > pos; i--) {
    arr[i] = arr[i - 1]; // Shift elements to the right
}
arr[pos] = val;
n++;

for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
}
```

Output:

```
1 2 99 3 4 5
```

Note:

In C++, when taking input using `cin`, **whitespaces act as separators**. For example:

```
string name;
```

```
cin >> name;
```

If you type `John Doe`, only `John` will be stored.
To take a full line including spaces, use:

```
getline(cin, name);
```

Deletion

Definition: Removing an element from a specific index.

Code:

```
int arr[10] = {1, 2, 3, 4, 5};

int n = 5;

int pos = 2; // index to delete (value = 3)


for (int i = pos; i < n - 1; i++) {

    arr[i] = arr[i + 1]; // Shift elements to the left

}

n--;


for (int i = 0; i < n; i++) {

    cout << arr[i] << " ";

}
```

Output:

```
1 2 4 5
```

Searching

Definition: Finding whether a value exists in the array and where.

Code:

```
int arr[] = {10, 20, 30, 40, 50};

int key = 30;

int n = 5;

bool found = false;


for (int i = 0; i < n; i++) {

    if (arr[i] == key) {

        cout << "Found at index " << i;

        found = true;

        break;

    }

}

if (!found) {

    cout << "Element not found.";
}
```

Disadvantages of Arrays:

Fixed Size: Once declared, the size of an array cannot be changed. This can lead to wasted memory (if the array is too large) or insufficient space (if it's too small).

Insertion & Deletion are Costly: Adding or removing elements (especially in the middle) requires shifting elements, which is time-consuming.

Homogeneous Data Only: Arrays can only store data of one type (e.g., all integers or all strings), limiting flexibility.

No Built-in Bounds Checking: Accessing an invalid index doesn't throw an error by default in C++, which can lead to unexpected bugs or crashes.

Real-World Examples of Arrays:

- Student Report Cards – storing marks of each subject.
- Music Playlists – storing song titles in order.
- To-do Lists – tracking tasks one by one.
- Temperature Logs – recording hourly sensor data.
- Game Scores – saving the top 10 scores in a leaderboard.

Problem Statement:

Write a C++ program to shift all elements of an array one position to the right. The last element should become the first element. Take the elements from the user.

The solution to this problem would be there in the next PDF, where we'll cover a better version (and the most commonly used Data Type in C++) of arrays known as **Vectors.**