Arrays and Vectors

Arrays and vectors let you store multiple values in a single variable. They are useful when dealing with groups of data—like a list of scores or names.

1 Arrays

An array is a fixed-size collection of elements of the same type. Each element is accessed by an index (starting from 0).

```
// Declare and initialize an array of size 5
int scores[5] = {90, 85, 78, 92, 88};

// Access and modify elements
cout << "First score: " << scores[0] << endl;
scores[1] = 95;
```

Key Points:

- Array size must be known at compile time.
- Indices start from 0 and go up to size - 1.
- Accessing an invalid index leads to undefined behavior.

Memory Layout:

An array stores elements in **contiguous memory locations**. This means if `scores[0]` is at address `1000` and each `int` takes 4 bytes, then:

- `scores[1]` is at address `1004`
- `scores[2]` is at address `1008`
- and so on

This layout allows fast access to any element using its index and is important for performance in loops.

2 Looping Over Arrays

Arrays work well with loops to process each element.

```
for (int i = 0; i < 5; i++) {
    cout << "Score " << i << ": " << scores[i] << endl;
}
```

3 Multidimensional Arrays

You can have arrays of arrays, useful for tables or grids.

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};

cout << matrix[1][2];  // Outputs 6
```

4 Vectors (C++ Standard Library)

A `vector` is a dynamic array. Unlike regular arrays, vectors can grow or shrink in size during runtime.

To use vectors, include the header:

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

Basic Usage:

```
vector<int> numbers;          // empty vector
numbers.push_back(10);        // add 10
numbers.push_back(20);
cout << numbers[0];           // outputs 10
```

`<int>` means this vector will store elements of type `int`. You can replace `int` with other types like `double`, `char`, or `string`.

Accessing Elements Safely:

```
cout << numbers.at(1); // safer than numbers[1], checks bounds
```

The method `at(index)` throws an exception if the index is out of bounds, unlike the `[]` operator.

Removing the Last Element:

```
numbers.pop_back(); // removes the last element
```

This is useful when treating the vector like a stack (LIFO).

Understanding Iterators:

An iterator is like a pointer that allows you to traverse through elements in a container (like a vector). The begin() method returns an iterator pointing to the first element, and end() returns an iterator one past the last element.

```
for (vector::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    cout << *it << " ";
}
```

Here, `*it` accesses the value pointed to by the iterator.

Erasing Elements:

```
numbers.erase(numbers.begin() + 1); // removes the second element
```

The `erase()` function removes an element or a range of elements. You provide it an iterator pointing to the element(s) to remove.

Inserting Elements:

```
numbers.insert(numbers.begin() + 1, 15); // insert 15 at index 1
```

The `insert()` function adds an element at a specific position, again using an iterator.

Example - Full Workflow:

```
vector numbers = {10, 20, 30};
numbers.insert(numbers.begin() + 1, 15); // 10, 15, 20, 30
numbers.erase(numbers.begin() + 2);      // 10, 15, 30
numbers.pop_back();                      // 10, 15
for (int n : numbers) {
    cout << *it << " ";
}
```

Looping with Range-Based for (C++11):

```
for (int n : numbers) {
    cout << n << " ";
}
```

Why Vectors?

- Size can change at runtime.
- Provides useful functions like `push_back()`, `size()`, and `clear()`.
- Safer and more flexible than raw arrays.