Programming Fundamentals

Muhammad Ateeq

[Updated: 01-04-2023]

Vectors

So we can iterate over arrays using loops and indexes. Is there any direct way to iterate without using explicit indexes?

The enhanced for loop, also known as the range-based for loop, is a feature introduced in C++11 that allows you to iterate over the elements of a container, such as an array or a vector, without having to use iterators or indices. Here is an example of how to use the enhanced for loop to iterate over an array:

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {1, 2, 3, 4, 5};

    // Iterate over the array using the enhanced for loop
    for (auto num : arr) {
        cout << num << " "; // output: 1 2 3 4 5
    }
    return 0;
}
```

In this example, we declare an array of integers called arr with 5 elements. We then use the enhanced for loop to iterate over the elements of the array and output their values. Note that the syntax of the enhanced for loop is for (auto element : container), where element is a variable that represents each element of the container, and container is the container being iterated over. The auto keyword tells the compiler to deduce the type of the element.

Basics of Vectors

What are vectors?

In C++, a vector is a **container class** [the word container and class, both will not be introduced during this course. So don't worry until you get to know both these words better during your next course on Object-Oriented Programming] that is used to store a dynamic array of elements. It provides a flexible and efficient way to store and manipulate collections of data.

Before moving on, can you explain what does "dynamic" mean?

In C++, a vector is said to be dynamic because its size can be changed dynamically at runtime. This means that you can add or remove elements from the vector as needed, and the size of the vector will automatically adjust to accommodate the changes.

Vectors are similar to arrays, but they offer several advantages over arrays, including the ability to automatically resize themselves when elements are added or removed. This makes them particularly useful when the size of the collection is not known in advance. In summary, the size of array must be specified at the time of declaration and cannot be changed during execution of the program. The only way to change the size of the array is to stop the program, changes the size of array, recompile and then execute it. Whereas, in case of vectors the size can be increased or decreased during execution of the program without having to stop, change the size, recompile and execute steps. This makes vectors flexible and efficient in terms of memory usage.

And what is a container?
In C++, a container is a data structure that holds a collection of elements. The standard library provides several container classes, such as vector, list, set, map, and deque, which have different characteristics and are optimized for different use cases. An array is also a container in C++.

How to use vectors in C++?
To use vectors in C++, you need to include the <vector> header file. Here's an example of how to create a vector of integers:

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

int main() {
    int mera_array[100];
    vector<int> myVector; // create an empty vector of integers
    myVector.push_back(10);    // add an element to the end of the vector
    myVector.push_back(20);
    myVector.push_back(30);

    cout << "Size of vector: " << myVector.size() << endl;
    // output: Size of vector: 3
    cout << "Elements of vector: ";
    for (int i = 0; i < myVector.size(); i++) {
        cout << myVector[i] << " "; // output: Elements of vector: 10 20 30
    }

    return 0;
}
```

In this example, we create an empty vector of integers called myVector. We then add three elements to the end of the vector using the push_back() method. Finally, we output the size and contents of the vector using the size() method and a for loop.

How does the advantage of dynamic memory come into play here?
When we declare an array, we must know the limit of elements that we may need to store. If we think that we may have to store 10-20 elements usually but in extreme cases we may need to strore 100 elements, we would prefer to declare an array of size 100. Now, id the type of array is integer, 100*4 = 400 bytes will be reserves as long as the program executes not matter how much of it is actually used. In case of vectors, no memory is reserved until an element is added to vector (e.g., using push_back() or insert() methods), whereas the memory is freed/released (e.g., using pop_back() or erase() methods).

How are vectors created, initialized, and accessed?
Here are examples of how to create, initialize, and access vectors in C++:

Creating a Vector:
To declare a vector in C++, you first need to include the <vector> header file. Then you can declare a vector using the following syntax:

```
vector<type> vectorName;
```

Here, type is the type of the elements you want to store in the vector, and vectorName is the name you want to give to the vector. For example, if you want to create a vector of integers called myVector, you would use the following declaration:

```
vector<int> myVector;
```

This creates an empty vector of integers called myVector. You can add elements to the vector using the push_back() method, as shown in the previous example. You can also declare a vector with initial values using the following syntax:

```
vector<type> vectorName {val1, val2, val3, ...};
```

For example, if you want to create a vector of strings with initial values, you would use the following declaration:

```
vector<string> myStrings {"hello", "world", "!"};
```

This creates a vector of strings called myStrings with three initial values: "hello", "world", and "!".

Initializing a Vector:
In C++, you can initialize a vector in several ways. Here are some examples:

Initializing a vector with default values of a certain size

```
// Initialize a vector of 5 integers, all with default value of 0
vector<int> myVector(5);
```

Initializing a vector with a specific set of values

```
// Initialize a vector of integers with specific values
vector<int> myVector {1, 2, 3, 4, 5};
```

Initializing a vector with a range of values

```
// Initialize a vector of integers with a range of values
vector<int> myVector(10);
int value = 1;
for (auto& num : myVector) {
      num = value;
      value++;
}
for (auto num : myVector) {
      std::cout << num << " "; // output: 1 2 3 4 5 6 7 8 9 10
}
```

In this example, we initialize a vector of 10 integers with default value of 0, then use a loop to assign each element a value from 1 to 10. We then output the contents of the vector using another loop.

Accessing Elements of a Vector:
You can access elements of a vector in C++ using two ways.
1. Using the subscript operator [], which allows you to access an element at a specific index. Here are some examples:

```
vector<int> myVector {1, 2, 3, 4, 5};

// Access an element using the subscript operator
cout << myVector[2] << endl; // output: 3

// Modify an element using the subscript operator
myVector[3] = 10;
for (auto num : myVector) {
      cout << num << " "; // output: 1 2 3 10 5
}
```

In this example, we first declare a vector of integers called myVector with 5 elements. We then access the element at index 2 using the subscript operator and output its value. We also modify an element at index 3 using the subscript operator and then output the contents of the vector using a range-based for loop.

2. Another way is to use the at() method. The advantage of using at() method over subscripts is that it checks for the bound of the vector and does not allow accessing elements outside the size range of the vector. Here's an example of using the at() method:

```
vector<int> v {1, 2, 3, 4, 5};

// Accessing elements using at()
cout << "Element at index 0: " << v.at(0) << '\n'; // prints 1
cout << "Element at index 2: " << v.at(2) << '\n'; // prints 3

// Trying to access an out-of-bounds index using at()
try {
  cout << "Element at index 10: " << v.at(10) << '\n';
} catch (const out_of_range& e) {
  cerr << "Out of range exception: " << e.what() << '\n';
}
```

Ignore the use of try { } catch here. It is used for exception (runtime error) handling and you will get introduced to it later.

How to add/remove elements in vector? Is it possible to add/remove from any specific location?

To add or remove elements in a C++ vector, you can use the following methods:

Add-Method1: push_back() - To add an element to the end of the vector:

```
#include <vector>
#include <iostream>

int main() {
  vector<int> v {1, 2, 3};
  v.push_back(4);
  for(auto i: v) cout << i << ' '; // prints 1 2 3 4
}
```

Add-Method2: insert() - To insert an element at a specific position in the vector:

```
vector<int> v {1, 2, 3};
auto it = v.begin() + 1; // position to insert
```

```
v.insert(it, 4); // insert 4 at index 1
for(auto i: v) cout << i << ' '; // prints 1 4 2 3
```

Remove-Method1: pop_back() - To remove an element from the end of the vector:

```
vector<int> v {1, 2, 3};
v.pop_back(); // remove 3 from the end
for(auto i: v) cout << i << ' '; // prints 1 2
```

Remove-Method2: erase() - To remove an element from a specific position in the vector:

```
vector<int> v {1, 2, 3};
auto it = v.begin() + 1; // position to remove
v.erase(it); // remove element at index 1
for(auto i: v) cout << i << ' '; // prints 1 3
```

Note that when using erase() or insert(), the iterator passed as argument indicates the position where the operation should start. This iterator can be obtained using methods such as begin() or end(), or using arithmetic operators on existing iterators.

[Optional] Ok, but what is an iterator and what is its difference with indexes?
In C++, an iterator and an index are two different ways to access the elements of a container (such as an array, vector, or map).

An index is an integer value that represents the position of an element in the container. We can use the index to directly access an element in the container by specifying its position. For example, we can access the first element of a vector with the index 0.

An iterator, on the other hand, is an object that provides a way to access the elements of a container sequentially. Iterators allow us to move through the container and access its elements one by one, and they can be used to perform various operations on the elements.

Here are some key differences between iterators and indices in C++:

Indices are integers, while iterators are objects: Indices are simply integer values that represent the position of an element in the container. Iterators, on the other hand, are objects that provide a way to access the elements of the container.
Indices allow for random access, while iterators provide sequential access: With an index, we can directly access any element in the container by specifying its position. With an iterator, we can only access the elements sequentially by moving from one element to the next.
Indices can be used with arrays, while iterators are used with containers: We can use indices to access the elements of an array directly, while iterators are typically used with containers like vectors, lists, and maps.

Here's an example that demonstrates the use of indices and iterators to access the elements of a vector:

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

int main() {
```

```
    vector<int> numbers {1, 2, 3, 4, 5};

    // Using an index to access the third element of the vector
    cout << "The third element of the vector is: " << numbers[2] << '\n';

    // Using an iterator to iterate over the elements of the vector
    cout << "The elements of the vector are: ";
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
      cout << *it << ' ';
    }
    cout << '\n';

    return 0;
  }
```

In this example, we use the index 2 to access the third element of the vector (numbers[2]). We also use an iterator to iterate over the elements of the vector (numbers.begin() returns an iterator pointing to the first element, and numbers.end() returns an iterator pointing just past the end of the vector). Note that we use the * operator to dereference the iterator and access the value of the element it points to.