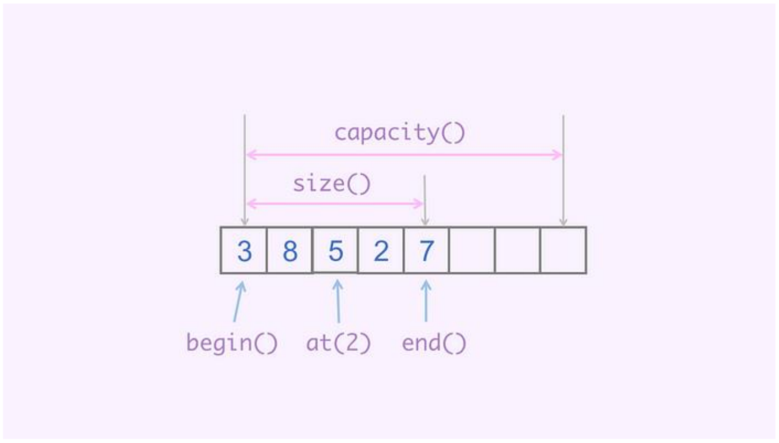
# Vector in C++



## Introduction

- Vectors are part of STL.
- Vectors are sequence containers representing arrays that can change in size.
- Vectors use **contiguous storage locations** for their elements, so their elements can also be accessed using **offsets** on regular pointers to their elements.
- Vector size can change dynamically, with their storage being handled automatically by the container.
- Vectors use a dynamically allocated array to store their elements.

## Internal Storage

- Vectors use a **dynamically allocated array** to store their elements.
- This array may need to be reallocated in order to grow in size when new elements are inserted (allocate a new array and move all elements to it) and it is expensive in terms of processing time.
- Vector containers may allocate some **extra storage** to accommodate for possible growth.
- Compared to arrays, vectors **consume more memory** in exchange for the ability to manage storage and grow dynamically.

## When to Use Vectors?

- Data is consistently changing.
- The size of data is unknown.
- Elements are not predefined.
- Compared to arrays, there are more ways to copy vectors.

## Declaration

Include `#include<vector>` library.

```
// Syntax: vector<object_type>

vector_variable_name;
#include <vector>
using namespace std;

vector<int> numbers;
```

## Initialization

1. Pushing the values one-by-one in vector using `push_back()`:

```
// Syntax: vector_name.push_back(element_value;

        numbers.push_back(3);
        numbers.push_back(8);
        // numbers = { 3, 8 }
```

2. Using the overload constructor of the vector Class:

```
// Syntax: vector<object_type> vector_name(number_of_repetition, element_value);

        vector<int> number1(2, 3);
        // numbers1 = { 3, 3 }
```

3. Using Array:

```
// Syntax: vector<object_type> vector_name {val1,val2,....,valn};

        vector<int> number2 { 9, 3, 6 };
        // numbers2 = { 9, 3, 6 }
```

4. Using already initialized vector:

```
// Syntax:
// vector<object_type> vector_name_2(vector_name_1.begin(),vector_name_1.end());

        vector<int> number3(number2.begin(), number2.end());

        // numbers3 = { 9, 3, 6 }
```

## Member Functions

### Iterators

- `begin()`: Returns an iterator pointing to the first element in the vector.
- `end()`: Returns an iterator pointing to the last element in the vector.

```
vector<int> a = { 3, 4, 5 };

for (auto i = a.begin(); i != a.end(); ++i) {
    cout << *i << " ";
}
cout << endl;
```

**Output:**

```
3 4 5
```

**Capacity**

- `size()`: Returns the number of elements currently present in the vector.
- `max_size()`: Returns the maximum number of elements that a vector can hold.
- `capacity()`: Returns the storage capacity currently allocated to the vector.
- `resize(n)`: Resizes the container to store `n` elements.
- `empty()`: Returns whether the container is empty or not.

```
vector<int> b = { 5, 7, 1 };

cout << b.size() << endl;
cout << b.max_size() << endl;
cout << b.capacity() << endl;

b.resize(5);
cout << b.size() << endl;
cout << b.empty() << endl;
```

**Output:**

```
3
2305843009213693951
3

5
0
```

*** *** ***

The cleanest way of iterating through a vector is via iterators:

```
for (auto it = begin (vector); it != end (vector); ++it) {
    it->doSomething ();
}
```

or (equivalent to the above)

```
for (auto & element : vector) {
    element.doSomething ();
}
```

## **Iterators vs. auto in C++**

The range-based `for` loop that is available since the C++ 11 standard, is a wonderful mean of making the code compact and clean. This post is totally dedicated to a single go: to show the difference between iterating using STL iterators and using `for` ranges.

Here's a simple code that creates a vector of integers and prints them all. The first program uses the old approach.

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

int main() {
    std::vector<int> data {10, 20, 30, 40};

    for (std::vector<int>::iterator i = data.begin(); i != data.end(); i++) {
        std::cout << *i << "\n";
    }
}
```

You can clearly see how heavy is the code. Of course, you can simplify it a bit with the `using namespace std` instruction or creating the `typedef` for the vector type. The latter is actually a good idea as that helps to avoid constructing the same data type twice.

C++ 11 provides use of **ranged-based `for` loops**.

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

int main() {
    std::vector<int> data {10, 20, 30, 40};

    for (auto x : data) {
        std::cout << x << "\n";
    }
}
```

Two things to note here.

First, there's no explicit iterator itself.

**Second, the `auto` keyword gives the compiler ability to deduce the correct type for the x variable.**

You—as a developer—do not need to think about that at the point of entering the loop. You already created your data and its type.

Also, notice the way vectors can be initialised in modern C++:

```
std::vector<int> data {10, 20, 30, 40};
```