Chapter 17 – STL Vectors in C++

A **vector** in C++ is a dynamic array that can grow or shrink in size automatically. It is part of the Standard Template Library (**STL**) and is defined in the <vector> header.

Advantages of Vectors

- **Dynamic sizing** (automatic resizing)
- **Efficient random access** (like arrays)
- **Memory management is handled automatically**
- **Provides various utility functions**

Common Methods of std::vector

Method	Description
push_back(value)	Adds an element to the end of the vector.
pop_back()	Removes the last element.
size()	Returns the number of elements in the vector.
capacity()	Returns the number of elements the vector can hold before resizing.
resize(n, value)	Resizes the vector to contain n elements. Optional: fill new elements with value.
clear()	Removes all elements from the vector.
empty()	Returns true if the vector is empty, false otherwise.
at(index)	Returns a reference to the element at a specific position (with bounds checking).
operator[]	Returns a reference to an element at a specific index (without bounds checking).
insert(iterator, value)	Inserts value at a specified position.
erase(iterator)	Removes an element at a specified position.
begin()	Returns an iterator to the first element.
end()	Returns an iterator to one past the last element.
front()	Returns a reference to the first element.
back()	Returns a reference to the last element.
swap(other_vector)	Swaps contents with another vector.
assign(n, value)	Replaces all elements with n copies of value.
shrink_to_fit()	Reduces capacity to fit the current size.

Examples

1. Basic Vector Operations

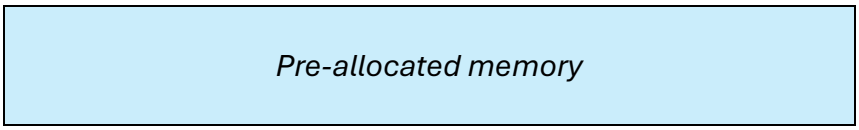
```
//Generic function to print a caption and a vector
template <typename T>
void showVector(const vector<T>& v, string caption = " ") {
    cout << caption;
    for (T i : v) {
        cout << i << " ";
    }
    cout << endl;
}
```

```
void experiment01()
{
    // Create an empty vector
    vector<int> numbers;

    // Adding elements
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    showVector(numbers, "Vector elements: ");

    // Accessing elements
    cout << "\nFirst element: " << numbers.front();
    cout << "\nLast element: " << numbers.back();
    cout << "\nElement at index 1: " << numbers.at(1);
    cout << "\nElement at index 1: " << numbers[1];

    // Removing the last element
    numbers.pop_back();
    cout << "\nAfter pop_back, last element: " << numbers.back() << endl;
    showVector(numbers, "Vector elements after pop_back: ");
}
```

```
Vector elements: 10 20 30
First element: 10
Last element: 30
Element at index 1: 20
Element at index 1: 20
After pop_back, last element: 20

All done!
```

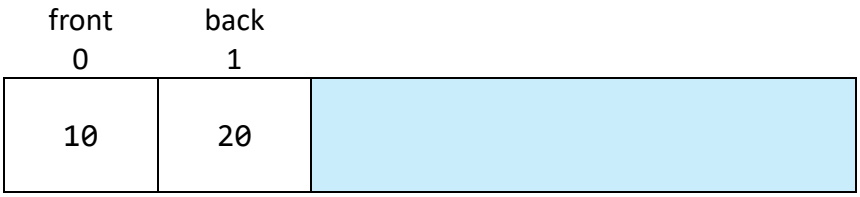
```
vector<int> numbers;
```

size = 0 (no data yet)

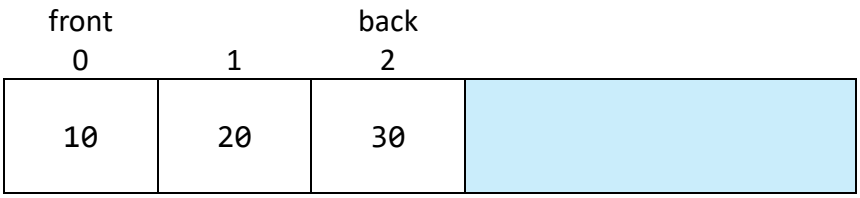



```
numbers.push_back(10);
```


```
numbers.push_back(20);
```




```
numbers.push_back(30);
```

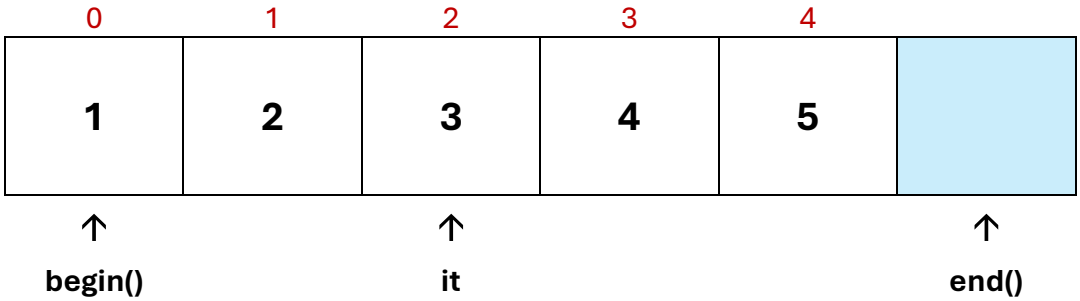


2. Using Iterators

```
void experiment02() {
    vector<int> vec = { 1, 2, 3, 4, 5 };

    cout << "Vector elements using iterators: ";

    for (auto it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

}
```

Output:

```
Vector elements using iterators: 1 2 3 4 5
```



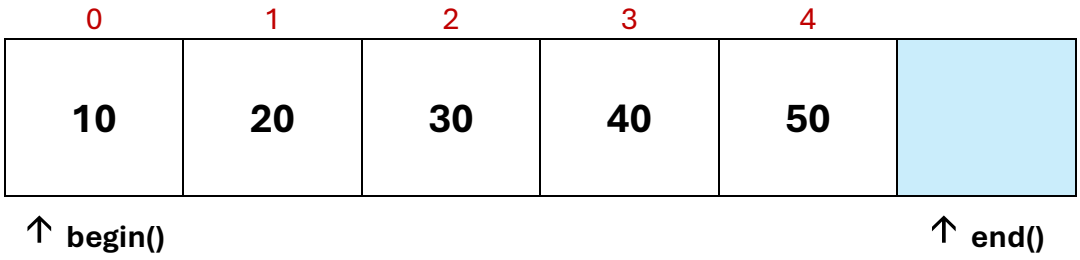
3. Inserting and Erasing Elements

```
void experiment03() {
    vector<int> vec = { 10, 20, 30, 40, 50 };
    showVector(vec, "Vector before insert and erase:        ");

    // Insert 25 at position 2 (0-based index)
    vec.insert(vec.begin() + 2, 25);
    showVector(vec, "Vector after insert 25 at position 2:  ");

    // Erase element at position 4
    vec.erase(vec.begin() + 4);
    showVector(vec, "Vector after erase cell at position 4: ");

}
```

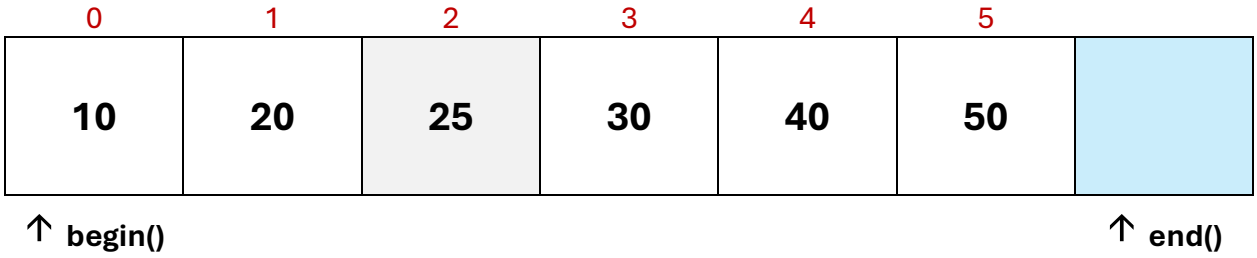
Output:

```
Vector before insert and erase:        10 20 30 40 50
Vector after insert 25 at position 2:  10 20 25 30 40 50
Vector after erase cell at position 4: 10 20 25 30 50
```

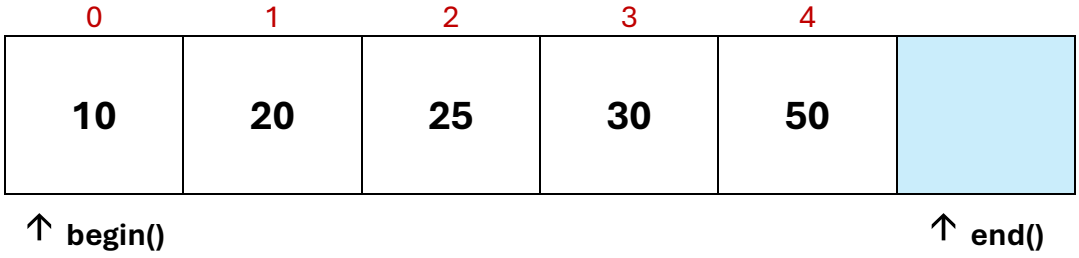
```
vector<int> vec = { 10, 20, 30, 40, 50 };
```




```
vec.insert(vec.begin() + 2, 25);
```




```
vec.erase(vec.begin() + 4);
```



4. Sorting a Vector

```
void experiment04() {
    vector<int> numbers = { 50, 10, 40, 20, 30 };
    showVector(numbers, "Original vector: ");

    sort(numbers.begin(), numbers.end());

    showVector(numbers, "Sorted vector:   ");
}
```

Output:

```
Original vector: 50 10 40 20 30
Sorted vector:   10 20 30 40 50
```

5. Using resize() and shrink_to_fit()

```
void experiment05() {
    vector<int> vec(5, 100); // Initialize vector with 5 elements, each 100
    showVector(vec, "Vector elements: ");
    cout << "Size before resize:     " << vec.size() << endl;
    cout << "Capacity before resize: " << vec.capacity() << endl;

    vec.resize(3); // Reduce size to 3 elements

    cout << "Size after resize:      " << vec.size() << endl;
    cout << "Capacity after resize:  " << vec.capacity() << endl;

    vec.shrink_to_fit(); // Reduce capacity to fit size
    cout << "Size after shrink:      " << vec.size() << endl;
    cout << "Capacity after shrink:  " << vec.capacity() << endl;

}
```

Output:

```
Vector elements: 100 100 100 100 100
Size before resize:     5
Capacity before resize: 5
Size after resize:      3
Capacity after resize:  5
Size after shrink:      3
Capacity after shrink:  3
```

Conclusion

- **Vectors** provide a **dynamic** and **efficient** way to manage collections of data in C++.
- They offer **various methods** to modify, access, and iterate over elements.
- **Using iterators** and **sorting functions** makes vector manipulation more powerful.
- Always prefer vectors over raw arrays **unless performance/memory constraints demand otherwise**.