# C++ Vectors and STL Cheatsheet

## Introduction to Vectors

A `vector` in C++ is like a resizable array. Both vectors and arrays are data structures used to store multiple elements of the same data type. The key difference is that arrays have a fixed size, while vectors can dynamically grow or shrink as needed.

To use a vector, you must include the `<vector>` header file:

```
#include <vector>
```

## Creating a Vector

To create a vector, use the `vector` keyword, specify the data type it will store within angle brackets ( `<>` ), and then provide the vector's name. For example:

```
vector<string> cars;
```

You can also initialize a vector with elements at the time of declaration, similar to arrays:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Print vector elements
for (string car : cars) {
  cout << car << "\n";
}
```

**Note:** The data type of the vector cannot be changed after it has been declared.

## Accessing Vector Elements

You can access vector elements using their index number inside square brackets ( `[]` ). Vectors are 0-indexed.

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars[0];  // Outputs Volvo

// Get the second element
cout << cars[1];  // Outputs BMW
```

The `.front()` and `.back()` functions provide convenient ways to access the first and last elements, respectively:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars.front();

// Get the last element
cout << cars.back();
```

For accessing elements at a specified index, the `.at()` function is often preferred over `[]` because it provides bounds checking and throws an error if the index is out of range.

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the second element
cout << cars.at(1);

// Attempt to access an out-of-range element (throws an error)
// cout << cars.at(6);
```

## Modifying Vector Elements

To change the value of a specific element, you can use either the index number or the `.at()` function:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Change the value of the first element using index
cars[0] = "Opel";
cout << cars[0];  // Now outputs Opel

// Change the value of the first element using .at()
cars.at(0) = "Saab";
cout << cars.at(0);  // Now outputs Saab
```

## Adding and Removing Vector Elements

Vectors can grow and shrink dynamically. The `.push_back()` function adds an element to the end of the vector:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars.push_back("Tesla");
cars.push_back("VW");
```

The `.pop_back()` function removes the last element from the vector:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars.pop_back(); // Removes "Mazda"
```

**Note:** For frequent additions or removals from both ends, a `deque` might be a more suitable data structure.

## Vector Size and Emptiness

To get the number of elements in a vector, use the `.size()` function:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.size();  // Outputs 4
```

The `.empty()` function checks if the vector is empty, returning `1` (true) if it is, and `0` (false) otherwise:

```
vector<string> emptyCars;
cout << emptyCars.empty();  // Outputs 1 (true)

vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.empty();  // Outputs 0 (false)
```

## Iterating Through a Vector

You can loop through vector elements using a `for` loop with `.size()`:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.size(); i++) {
  cout << cars[i] << "\n";
}
```

Alternatively, a more readable range-based `for` loop (introduced in C++11) can be used:

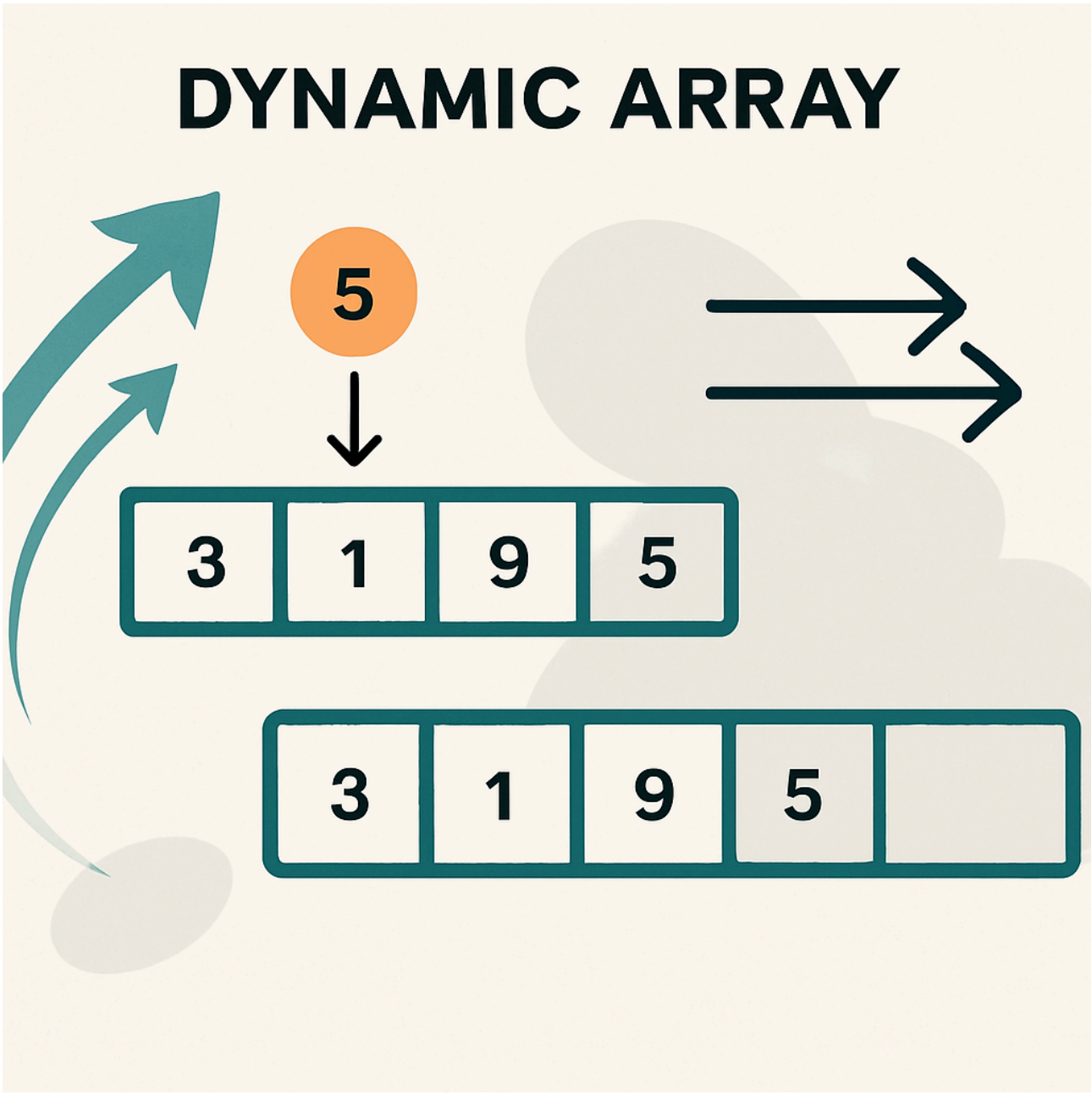
```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (string car : cars) {
  cout << car << "\n";
}
```

## Other Important STL Vector Functions

While the above covers the basics, `std::vector` offers many more functionalities. Here are a few notable ones:

- `clear()`: Removes all elements from the vector.
- `capacity()`: Returns the size of the storage space currently allocated for the vector, expressed in terms of elements.
- `reserve(n)`: Requests that the vector capacity be at least `n`.
- `resize(n)`: Resizes the container so that it contains `n` elements.
- `insert(position, value)`: Inserts `value` at `position`.
- `erase(position)`: Removes the element at `position`.

For a complete reference, refer to the official C++ documentation or resources like cppreference.com.




## Deep Dive into `std::vector` as a Data Structure

At its core, `std::vector` is a dynamic array. This means it manages a contiguous block of memory where its elements are stored. When the vector needs to grow beyond its current capacity, it typically allocates a new, larger block of memory, copies all existing elements to the new location, and then deallocates the old memory block. This reallocation process can be expensive, but `std::vector` employs a growth strategy (often doubling its capacity) to ensure that `push_back` operations have an amortized constant time complexity.

## Memory Management and Capacity

`std::vector` distinguishes between its `size()` (the number of elements currently stored) and its `capacity()` (the total number of elements it can hold without reallocating memory).

- **`size()`**: Returns the number of elements in the vector. This is the actual count of elements you have added.
- **`capacity()`**: Returns the size of the storage space currently allocated for the vector, expressed in terms of elements. This is the maximum number of elements the vector can hold before it needs to reallocate.
- **`reserve(n)`**: Requests that the vector capacity be at least `n`. If `n` is greater than the current capacity, a reallocation occurs. This can be used to pre-allocate memory and avoid multiple reallocations if you know the approximate number of elements you will store.
- **`shrink_to_fit()` (C++11 onwards)**: Reduces the capacity of the vector to fit its size. This can be useful to free up unused memory, but it might involve a reallocation.

### Example of Capacity and Resizing:

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

int main() {
    std::vector<int> v;
    std::cout << "Initial size: " << v.size() << ", capacity: " << v.capacity()
<< "\n";

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
        std::cout << "After push_back " << i << ": size: " << v.size() << ",
capacity: " << v.capacity() << "\n";
    }

    v.reserve(20); // Request capacity of at least 20
    std::cout << "After reserve(20): size: " << v.size() << ", capacity: " <<
v.capacity() << "\n";

    v.shrink_to_fit(); // Reduce capacity to fit size
    std::cout << "After shrink_to_fit(): size: " << v.size() << ", capacity: "
<< v.capacity() << "\n";

    return 0;
}
```

### Time Complexities of Common Operations

Understanding the time complexities is crucial for efficient program design.

| Operation | Time Complexity | Notes |
|---|---|---|
| Access element by index | O(1) | Direct memory access |
| `push_back()` | Amortized O(1) | Occasional reallocations make it O(N) in worst case, but average is O(1) |
| `pop_back()` | O(1) | Removing the last element is fast |
| Insert/Erase in middle | O(N) | Requires shifting subsequent elements |
| Insert/Erase at beginning | O(N) | Requires shifting all elements |
| `clear()` | O(N) | Destroys elements, but capacity usually remains |
| `resize()` | O(N) | May involve construction/destruction or reallocation |
| `reserve()` | O(N) | If reallocation occurs, elements are copied |

## `std::vector` and Iterators

Iterators are a fundamental concept in STL, providing a generic way to access elements of containers. `std::vector` provides random-access iterators, meaning you can move them by an arbitrary number of positions in constant time.

- **`begin()`**: Returns an iterator to the first element.
- **`end()`**: Returns an iterator to the element past the last element (a sentinel value).
- **`cbegin()`** / **`cend()`**: Return constant iterators (C++11 onwards), useful when you want to ensure elements are not modified.

- **`rbegin()` / `rend()`**: Return reverse iterators (C++11 onwards), allowing iteration from the last element to the first.

**Example of using Iterators:**

```
 #include <iostream>
#include <vector>
#include <algorithm> // For std::sort

int main() {
    std::vector<int> numbers = {5, 2, 8, 1, 9};

    // Iterate using traditional for loop with iterators
    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end();
++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";

    // Use range-based for loop (syntactic sugar for iterators)
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Sort the vector using iterators and std::sort
    std::sort(numbers.begin(), numbers.end());
    std::cout << "Sorted numbers: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

## Constructors and Initialization

`std::vector` offers several ways to construct and initialize a vector:

- **Default Constructor:** `std::vector<T> v;` (creates an empty vector).
- **Fill Constructor:** `std::vector<T> v(count, value);` (creates a vector with `count` elements, each initialized to `value`).
- **Range Constructor:** `std::vector<T> v(first, last);` (creates a vector with elements from the range `[first, last)`).
- **Copy Constructor:** `std::vector<T> v2(v1);` (creates a copy of an existing vector).

- **Move Constructor (C++11 onwards):** `std::vector<T> v2(std::move(v1));` (moves resources from `v1` to `v2`, leaving `v1` in a valid but unspecified state).
- **Initializer List Constructor (C++11 onwards):** `std::vector<T> v = {e1, e2, e3};` (initializes with a list of elements).

**Example:**

```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
    // Default constructor
    std::vector<int> v1;

    // Fill constructor
    std::vector<std::string> v2(5, "hello"); // 5 elements, all "hello"

    // Range constructor (from an array)
    int arr[] = {1, 2, 3, 4, 5};
    std::vector<int> v3(arr, arr + 5);

    // Copy constructor
    std::vector<int> v4 = v3;

    // Initializer list constructor
    std::vector<double> v5 = {1.1, 2.2, 3.3};

    std::cout << "v1 size: " << v1.size() << "\n";
    std::cout << "v2 elements: ";
    for (const std::string& s : v2) std::cout << s << " ";
    std::cout << "\n";
    std::cout << "v3 elements: ";
    for (int x : v3) std::cout << x << " ";
    std::cout << "\n";
    std::cout << "v4 elements: ";
    for (int x : v4) std::cout << x << " ";
    std::cout << "\n";
    std::cout << "v5 elements: ";
    for (double x : v5) std::cout << x << " ";
    std::cout << "\n";

    return 0;
}
```

## Comparison with Raw Arrays

While `std::vector` is often referred to as a dynamic array, it offers significant advantages over C-style raw arrays:

| Feature | `std::vector` | Raw C-style Array ( `int arr[]` ) |
|---|---|---|
| **Size Management** | Dynamic; automatically resizes | Fixed size at compile time or allocation |
| **Memory Safety** | Provides bounds checking ( `.at()` ), reduces risk of buffer overflows | No inherent bounds checking; prone to buffer overflows |
| **Functionality** | Rich set of member functions ( `push_back` , `pop_back` , `size` , `empty` , iterators, etc.) | Basic pointer arithmetic; requires manual management |
| **Memory Management** | Automatic allocation and deallocation | Manual allocation ( `new[]` ) and deallocation ( `delete[]` ) required |
| **Ease of Use** | Easier and safer to use | More error-prone, requires careful manual handling |

In modern C++, `std::vector` is almost always preferred over raw arrays for dynamic collections of elements due to its safety, convenience, and rich feature set. Raw arrays are typically used only in very specific performance-critical scenarios or when interfacing with C libraries.