Comprehensive Guide to Vectors in C++

C++ Programming Guide

December 27, 2025

Abstract

This document provides a comprehensive guide to using vectors in C++. Vectors are dynamic arrays that can resize themselves automatically when elements are added or removed. We cover declaration, initialization, adding/removing elements, and essential vector functions with practical examples.

Contents

1 Introduction, Declaration, and Initialization

1.1 What is a Vector?

A vector in C++ is a sequence container that represents a dynamic array. Unlike regular arrays, vectors can change their size dynamically, with their storage being handled automatically by the container.

1.2 Including the Vector Header

To use vectors, you must include the vector header:

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

1.3 Declaration and Initialization

Vectors can be declared and initialized in several ways:

```
// Method 1: Empty vector
std::vector<int> vec1;

// Method 2: Vector with initial size
std::vector<int> vec2(5);  // Vector with 5 elements, all initialized to 0

// Method 3: Vector with initial size and value
std::vector<int> vec3(5, 10);  // Vector with 5 elements, all initialized
    to 10

// Method 4: Initializer list (C++11 and later)
std::vector<int> vec4 = {1, 2, 3, 4, 5};

// Method 5: Copy from another vector
std::vector<int> vec5(vec4);

// Method 6: From array
int arr[] = {10, 20, 30, 40, 50};
std::vector<int> vec6(arr, arr + 5);
```

Example: Basic Vector Operations

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

int main() {
    // Declaration and initialization
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Accessing elements
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Second element: " << numbers.at(1) << std::endl;

    // Display all elements
    std::cout << "All elements: ";
    for(int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

```
First element: 1
Second element: 2
All elements: 1 2 3 4 5
```

2 Adding Elements with push_back() Method

2.1 The push_back() Method

The `push_back()` method adds a new element at the end of the vector, after its current last element. This increases the vector size by one.

2.2 Syntax

```
vector_name.push_back(value);
```

Example: Using push_back()

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

int main() {
    std::vector<int> numbers;

    std::cout << "Initial vector size: " << numbers.size() << std::
    endl;

    // Adding elements using push_back()
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);
    numbers.push_back(40);
    numbers.push_back(50);

    std::cout << "After adding elements, vector size: "
              << numbers.size() << std::endl;

    // Display all elements
    std::cout << "Vector elements: ";
    for(int i = 0; i < numbers.size(); i++) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;

    // Add more elements
    numbers.push_back(60);
    numbers.push_back(70);

    std::cout << "Final vector size: " << numbers.size() << std::endl;
    std::cout << "Final vector elements: ";
    for(int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

```
Initial vector size: 0
After adding elements, vector size: 5
Vector elements: 10 20 30 40 50
Final vector size: 7
Final vector elements: 10 20 30 40 50 60 70
```

3 Vector of Structs Basics

3.1 Using Structs with Vectors

Vectors can store user-defined types like structs. This is particularly useful for creating collections of complex data.

Example: Vector of Structs

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

// Define a struct
struct Student {
    std::string name;
    int age;
    double grade;
};

int main() {
    // Create a vector of Student structs
    std::vector<Student> students;

    // Create student objects
    Student s1 = {"Alice", 20, 85.5};
    Student s2 = {"Bob", 21, 78.3};
    Student s3 = {"Charlie", 22, 92.1};

    // Add students to vector using push_back()
    students.push_back(s1);
    students.push_back(s2);
    students.push_back(s3);

    // Add a student directly
    students.push_back({"Diana", 19, 88.7});

    // Display all students
    std::cout << "Student Records:" << std::endl;
    std::cout << "----------------" << std::endl;

    for(const auto& student : students) {
        std::cout << "Name: " << student.name
                  << ", Age: " << student.age
                  << ", Grade: " << student.grade
                  << std::endl;
    }

    // Access and modify a specific student
    students[1].grade = 80.0;  // Update Bob's grade

    std::cout << "\nAfter updating Bob's grade:" << std::endl;
    std::cout << "Bob's new grade: " << students[1].grade << std::endl
    ;

    return 0;
}
```

Output:

```
Student Records:
----------------
Name: Alice, Age: 20, Grade: 85.5
Name: Bob, Age: 21, Grade: 78.3
Name: Charlie, Age: 22, Grade: 92.1
Name: Diana, Age: 19, Grade: 88.7

After updating Bob's grade:
Bob's new grade: 80
```

4 Removing Elements with pop_back() Method

4.1 The pop_back() Method

The `pop_back()` method removes the last element in the vector, reducing the container size by one.

4.2 Important Notes

- `pop_back()` removes the last element but doesn't return it
- The element is destroyed
- Calling `pop_back()` on an empty vector results in undefined behavior
- Always check if the vector is empty before using `pop_back()`

Example: Using pop_back()

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

int main() {
    std::vector<int> numbers = {10, 20, 30, 40, 50, 60, 70};

    std::cout << "Original vector: ";
    for(int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\nSize: " << numbers.size() << std::endl;

    // Remove last element
    numbers.pop_back();

    std::cout << "\nAfter first pop_back(): ";
    for(int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\nSize: " << numbers.size() << std::endl;

    // Remove two more elements
    numbers.pop_back();
    numbers.pop_back();

    std::cout << "\nAfter two more pop_back(): ";
    for(int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\nSize: " << numbers.size() << std::endl;

    // Safe removal with empty check
    while(!numbers.empty()) {
        std::cout << "\nRemoving: " << numbers.back();
        numbers.pop_back();
    }

    std::cout << "\n\nFinal vector size: " << numbers.size() << std::
    endl;

    // Check before popping from empty vector
    if(!numbers.empty()) {
        numbers.pop_back();  // Safe
    } else {
        std::cout << "Vector is empty, cannot pop_back()" << std::endl
    ;
    }

    return 0;
}
```

Output:

```
Original vector: 10 20 30 40 50 60 70
Size: 7

After first pop_back(): 10 20 30 40 50 60
Size: 6

After two more pop_back(): 10 20 30 40
Size: 4
```

5 Essential Vector Functions

5.1 Core Vector Functions

Vectors provide several member functions for accessing and managing elements:

5.1.1 `front()` and `back()`

- `front()`: Returns a reference to the first element
- `back()`: Returns a reference to the last element

5.1.2 `empty()` and `size()`

- `empty()`: Returns true if the vector is empty
- `size()`: Returns the number of elements in the vector

5.1.3 `capacity()` and `reserve()`

- `capacity()`: Returns the size of allocated storage capacity
- `reserve(n)`: Requests that the vector capacity be enough to contain n elements

5.1.4 `clear()` and `resize()`

- `clear()`: Removes all elements from the vector
- `resize(n)`: Resizes the container to contain n elements

Example: Essential Vector Functions

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

int main() {
    std::vector<int> numbers = {10, 20, 30, 40, 50};

    // front() and back()
    std::cout << "First element (front()): " << numbers.front() << std
    ::endl;
    std::cout << "Last element (back()): " << numbers.back() << std::
    endl;

    // empty() and size()
    std::cout << "\nIs vector empty? "
              << (numbers.empty() ? "Yes" : "No") << std::endl;
    std::cout << "Vector size: " << numbers.size() << std::endl;

    // capacity()
    std::cout << "\nInitial capacity: " << numbers.capacity() << std::
    endl;

    // Demonstrate capacity growth
    std::vector<int> demo;
    std::cout << "\nDemonstrating capacity growth:" << std::endl;
    for(int i = 0; i < 10; i++) {
        demo.push_back(i);
        std::cout << "Size: " << demo.size()
                  << ", Capacity: " << demo.capacity() << std::endl;
    }

    // reserve()
    std::vector<int> reservedVec;
    reservedVec.reserve(100);
    std::cout << "\nAfter reserve(100):" << std::endl;
    std::cout << "Size: " << reservedVec.size()
              << ", Capacity: " << reservedVec.capacity() << std::endl
    ;

    // clear()
    std::vector<int> toClear = {1, 2, 3, 4, 5};
    std::cout << "\nBefore clear() - Size: " << toClear.size() << std
    ::endl;
    toClear.clear();
    std::cout << "After clear() - Size: " << toClear.size()
              << ", Empty? " << (toClear.empty() ? "Yes" : "No") <<
    std::endl;

    // resize()
    std::vector<int> toResize = {1, 2, 3};
    std::cout << "\nOriginal: ";
    for(int num : toResize) std::cout << num << " ";

    toResize.resize(5, 99);  // Resize to 5, new elements = 99
    std::cout << "\nAfter resize(5, 99): ";
    for(int num : toResize) std::cout << num << " ";

    toResize.resize(2);  // Shrink to 2 elements
    std::cout << "\nAfter resize(2): ";
    for(int num : toResize) std::cout << num << " ";
    std::cout << std::endl;

    // at() for bounds checking
```

6 Summary

6.1 Key Points

- Vectors are dynamic arrays that can resize automatically
- Use `push_back()` to add elements at the end
- Use `pop_back()` to remove the last element
- Always check if vector is empty before using `pop_back()`
- `front()` and `back()` provide access to first and last elements
- `size()` returns the number of elements, `capacity()` returns allocated space
- `reserve()` can pre-allocate memory for better performance
- Vectors can store user-defined types like structs

6.2 Best Practices

1. Use `reserve()` when you know the approximate number of elements to avoid reallocations
2. Use range-based for loops for cleaner iteration
3. Prefer `at()` over `[]` for bounds-checked access
4. Use `empty()` to check if vector is empty instead of comparing `size() == 0`
5. Consider using `emplace_back()` instead of `push_back()` for complex types (C++11+)

6.3 Common Use Cases

- Storing collections of data where size may change
- Implementing stacks (using `push_back()` and `pop_back()`)
- Collections of objects with different types using structs/classes
- Temporary storage for processing data