# Vector

## What Is std::vector?

### Definition

std::vector is a **dynamic array** that can resize itself automatically when elements are added or removed. It provides **random access** to its elements.

### Memory Management

* **Dynamic allocation**: Vector allocates memory dynamically on the **heap**.
* **Capacity**: It maintains a capacity, which is the size of the allocated storage. When the number of elements exceeds the capacity, it allocates a new larger array, copies the existing elements to the new array, and frees the old array. Note: Size is the number of elements currently stored, while capacity is the total number of elements that can be stored without reallocating.

### Similar Data Structures

* std::array: A fixed-size array
* std::list: A doubly linked list
* std::deque: A double-ended queue that allows insertion and deletion from both ends

## Pros and Cons

### Pros

* **Dynamic sizing**: When you add elements beyond its current capacity, it automatically allocates more memory and moves existing elements to the new storage.
* **Random access**: O(1).
* **Good cache locality**: Vector stores its elements in **contiguous memory** locations.
* **STL** **algorithm supports**: Such as std:sort, std:find, and std::for\_each.

### Cons

* **Costly insertion/deletion in the middle or at the front**: O(n) – All subsequent elements need to be **shifted to maintain order**.
* **May cause performance overhead**: Vector has to **reallocates** its memory when the number of elements exceeds the current capacity, which involves allocating new storage, copying existing elements to the new location, and freeing the old storage.

The maximum capability of a vector is represented by the type std::vector::size\_type, which is typically defined as std::size\_t (2^32 – 1 on 32-bit systems, or 2^64 - 1 on 64-bit systems).

There is std::vector::max\_size() that you can use to see the maximum number of elements (maximum capability / size of each item) the the vector you declared can potentially hold.

## Iteration

### Using Traditional For Loop

#include <iostream>

#include <vector>

int main() {

std::vector<int> vec = {1, 2, 3, 4, 5};

for (size\_t i = 0; i < vec.size(); ++i) {

std::cout << vec[i] << " ";

}

std::cout << std::endl;

return 0;

}

### Using Range-Based For Loop (C++11 and above)

#include <iostream>

#include <vector>

int main() {

std::vector<int> vec = {1, 2, 3, 4, 5};

for (int value : vec) { // You can modity to "for (const int& value : vec)" to avoid unnecessary copies

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

}

### Using Iterator

#include <iostream>

#include <vector>

int main() {

std::vector<int> vec = {1, 2, 3, 4, 5};

for (auto it = vec.begin(); it != vec.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl; // 1 2 3 4 5

return 0;

}

### Using Reverse Iterator

#include <iostream>

#include <vector>

int main() {

std::vector<int> vec = {1, 2, 3, 4, 5};

for (auto it = vec.rbegin(); it != vec.rend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl; // 5 4 3 2 1

return 0;

}

### Using std::for\_each Algorithm

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> vec = {1, 2, 3, 4, 5};

std::for\_each(vec.begin(), vec.end(), [](int value) {

std::cout << value << " ";

});

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Links** |
| push\_back | T value | Adds an element to the end of the vector. | O(1) | O(1) | vec.push\_back(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/push_back) |
| pop\_back | None | Removes the last element of the vector. | O(1) | O(1) | vec.pop\_back(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/pop_back) |
| insert | iterator pos,  T value | Inserts an element at the specified position. | O(n) | O(1) | vec.insert(  vec.begin() + 1, 20); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/insert) |
| erase | iterator pos | Removes the element at the specified position. | O(n) | O(1) | vec.erase(  vec.begin() + 1); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/erase) |
| clear | None | Removes all elements from the vector. | O(n) | O(1) | vec.clear(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/clear) |
| size | None | Returns the number of elements in the vector. | O(1) | O(1) | size\_t s = vec.size(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/size) |
| capacity | None | Returns the number of elements that the vector can hold without reallocating. | O(1) | O(1) | size\_t c = vec.capacity(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/capacity) |
| resize | size\_type count,  T value = T() | Resizes the vector to contain count elements. | O(n) | O(1) | vec.resize(10, 0); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/resize) |
| at | size\_type pos | Accesses the element at the specified position with bounds checking.  Bound checking = Throwing an std::out\_of\_range exception if the index is out of range. | O(1) | O(1) | int val = vec.at(2); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/at) |
| operator[] | size\_type pos | Accesses the element at the specified position without bounds checking. Accessing out of bounds leads to undefined behavior. | O(1) | O(1) | int val = vec[[2]]; | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/operator_at) |
| front | None | Returns a reference to the first element. | O(1) | O(1) | int first = vec.front(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/front) |
| back | None | Returns a reference to the last element. | O(1) | O(1) | int last = vec.back(); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/back) |
| swap | vector& other | Swaps the contents of the vector with another vector. | O(1) | O(1) | vec.swap(otherVec); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/swap) |
| assign | size\_type count,  const T& value | Assigns new contents to the vector, replacing its current contents. | O(n) | O(1) | vec.assign(5, 10); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/assign) |
| emplace\_back | Args&&... args | Constructs an element in-place at the end of the vector.  Better than push\_back.  In-place = create an object directly in memory, rather than creating it separately and then moving / copying it to that location. | O(1) | O(1) | vec.emplace\_back(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/emplace_back) |
| emplace | iterator pos,  Args&&... args | Constructs an element in-place at the specified position.  Better than insert. | O(n) | O(1) | vec.emplace(  vec.begin() + 1, 20); | [cplusplus](https://en.cppreference.com/w/cpp/container/vector/emplace) |
| data |  | Returns a pointer to the underlying array serving as element storage.  The pointer is in range [data(), data() + size()]. |  |  |  |  |

## Example

### Ex1: Quick Start

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  #include <algorithm>    int main() {      // Create a vector and add elements      std::vector<int> vec;      vec.push\_back(1);      vec.push\_back(2);      vec.push\_back(4);      vec.push\_back(5);        // Display size and capacity      std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;        // Insert an element at index 2      auto itr = vec.begin();      vec.insert(itr + 2, 3);        // Display elements using range-based for loop      std::cout << "Elements after insert(): ";      for (const auto& value : vec) {          std::cout << value << " ";      }      std::cout << std::endl;        // Remove the last element      vec.pop\_back();      // Remove element at index 2      vec.erase(vec.begin() + 2);        // Display elements using traditional for loop      std::cout << "Elements after pop\_back() and erase(): ";      for (int i = 0; i < vec.size(); i++) {          std::cout << vec[i] << " ";      }      std::cout << std::endl;        // Clear the vector      vec.clear();      std::cout << "Size after clear: " << vec.size() << std::endl;        // Refill the vector      vec = {1, 2, 3, 4, 5};        // Resize the vector and fill with 0      vec.resize(10, 0);        // Display elements using std::for\_each      std::cout << "Elements after resize: ";      std::for\_each(vec.begin(), vec.end(), [](int value) {          std::cout << value << " ";      });      std::cout << std::endl;        // Display first and last element      std::cout << "First element: " << vec.front() << ", last elemement: " << vec.back() << std::endl;        return 0;  } | Size: 4, Capacity: 4  Elements after insert(): 1 2 3 4 5  Elements after pop\_back(): 1 2 4  Size after clear: 0  Elements after resize: 1 2 3 4 5 0 0 0 0 0  First element: 1, last elemement: 0 |

### Ex2: Vector and Bytes

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  #include <cstring> // memcpy  #include <cstdint> // uint8\_t    int main() {  // Create a vector  std::vector<uint8\_t> vec;    // Initialize all items in vector to 0  std::fill(vec.begin(), vec.end(), 0U);    // Allocate memory for vector. Maximum is 5 items  int VEC\_SIZE = 5;  vec.resize(VEC\_SIZE); // NOTE: Without this, segmentation fault at memcpy()!    // Copy data to vector  uint8\_t buffer[] = {'a', 'b', 'c', 'x', 'y'};  std::memcpy(vec.data(), buffer, VEC\_SIZE);    // Display size and capacity  std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;    // Display elements using data buffer  std::cout << "Display buffer in vector: 0x";  const uint8\_t\* pbuffer = vec.data();  for (int i = 0; i < vec.size(); i++) {  printf("%02X", pbuffer[i]); // NOTE: uin8\_t cannot be used with std::count  }  std::cout << std::endl;    return 0;  } | Size: 5, Capacity: 5  Display buffer in vector: 0x6162637879 |

# Array

## What Is std::array?

### Definition

std::array is a container that encapsulates **fixed-size arrays**. It provides a safer and more **feature-rich alternative to C-style arrays**.

### Memory Management

* **Fixed Size**: The size of arrays is fixed at compile time and cannot be changed during runtime. This is different from dynamic containers like std::vector.
* **Static Allocation**: Memory for arrays is allocated on the **stack** (if declared locally) or in static storage (if declared globally or as static). This leads to faster allocation and deallocation compared to heap-allocated structures.
* **Contiguous Memory**: Elements in arrays are stored in **contiguous memory**, similar to C-style arrays, allowing efficient access.

### Similar Data Structures

* std::vector: A dynamic array that can grow and shrink in size at runtime.
* std::deque: A double-ended queue that allows fast insertions and deletions at both ends.
* **C-style arrays**: Traditional arrays in C and C++, but they lack the functionality and safety features of std::array.

## Pros and Cons

### Pros

* **Fixed Size**: Improve performance since the memory is allocated at **compile time**.
* **Type Safety**: Unlike C-style arrays, std::array is a **template** and provides type safety, preventing type-related errors.
* **Built-in Functions**: Provides a variety of member functions (e.g., size(), at()) that enhance functionality and safety.
* **Contiguous Storage**: Provides efficient access patterns, similar to traditional arrays.

### Cons

* **Fixed Size**: Cannot be resized at runtime, which limits flexibility compared to dynamic containers like std::vector. If you need a container that can grow, std::array is not suitable.
* **Slight Overhead**: There may be a slight performance overhead due to additional member functions compared to raw arrays.

## Iteration

### Traditional For Loop with Indexing

#include <iostream>

#include <array>

int main() {

std::array<int, 5> arr = {1, 2, 3, 4, 5};

for (size\_t i = 0; i < arr.size(); ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

return 0;

}

### Range-Based For Loop

#include <iostream>

#include <array>

int main() {

std::array<int, 5> arr = {1, 2, 3, 4, 5};

for (const auto& value : arr) {

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

}

### Using Iterators

#include <iostream>

#include <array>

int main() {

std::array<int, 5> arr = {1, 2, 3, 4, 5};

for (auto it = arr.begin(); it != arr.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Using std::for\_each

#include <iostream>

#include <array>

#include <algorithm>

int main() {

std::array<int, 5> arr = {1, 2, 3, 4, 5};

std::for\_each(arr.begin(), arr.end(), [](int value) {

std::cout << value << " ";

});

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| at | size\_type pos | Accesses the element at the specified position with bounds checking. | O(1) | O(1) | arr.at(1); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/at) |
| operator[] | size\_type pos | Accesses the element at the specified position without bounds checking. | O(1) | O(1) | arr[1]; | [cplusplus](https://en.cppreference.com/w/cpp/container/array/operator_at) |
| front | None | Returns a reference to the first element. | O(1) | O(1) | arr.front(); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/front) |
| back | None | Returns a reference to the last element. | O(1) | O(1) | arr.back(); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/back) |
| data | None | Returns a pointer to the underlying array. | O(1) | O(1) | arr.data(); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/data) |
| size | None | Returns the number of elements in the array. | O(1) | O(1) | arr.size(); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/size) |
| fill | const T& value | Assigns the given value to all elements of the array. | O(n) | O(1) | arr.fill(0); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/fill) |
| swap | array& other | Swaps the contents of the array with another array. | O(1) | O(1) | arr.swap(otherArr); | [cplusplus](https://en.cppreference.com/w/cpp/container/array/swap) |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <array>  #include <algorithm>    int main() {  // Create and initialize a std::array  std::array<int, 5> arr = {1, 2, 3, 4, 5};    // Access elements using at() and operator[]  std::cout << "Element at index 2 using at(): " << arr.at(2) << std::endl;  std::cout << "Element at index 2 using operator[]: " << arr[2] << std::endl;    // Access first and last elements  std::cout << "First element: " << arr.front() << std::endl;  std::cout << "Last element: " << arr.back() << std::endl;    // Size of the array  std::cout << "Size of array: " << arr.size() << std::endl;    // Fill the array  arr.fill(0);  std::cout << "Array after fill(0): ";  for (const auto& value : arr) {  std::cout << value << " ";  }  std::cout << std::endl;    // Swap with another array  std::array<int, 5> otherArr = {10, 20, 30, 40, 50};  arr.swap(otherArr);  std::cout << "Array after swap: ";  for (const auto& value : arr) {  std::cout << value << " ";  }  std::cout << std::endl;    // Using std::for\_each to print the array  std::cout << "Using for\_each to print elements: ";  std::for\_each(arr.begin(), arr.end(), [](int value) {  std::cout << value << " ";  });  std::cout << std::endl;    return 0;  } | Element at index 2 using at(): 3  Element at index 2 using operator[]: 3  First element: 1  Last element: 5  Size of array: 5  Array after fill(0): 0 0 0 0 0  Array after swap: 10 20 30 40 50  Using for\_each to print elements: 10 20 30 40 5 |

# List

## What Is std::list?

### Definition

std::list is a **doubly-linked list** container that allows for **efficient insertion and deletion** of elements from anywhere in the list. Unlike std::array or std::vector, it does not require contiguous memory allocation.

### Memory Management

* **Node-Based Structure**: Each element in a std::list is stored in a separate node, which contains **pointers** to both the previous and next nodes. This allows for **bidirectional traversal**.
* **Dynamic Allocation**: Nodes are allocated on the **heap**, which means memory can be dynamically allocated and deallocated as elements are added or removed.
* **Non-Contiguous Memory**: Since nodes are allocated individually, elements can be scattered throughout memory.

### Similar Data Structures

* std::vector: A dynamic array that allows random access and is stored in contiguous memory. It is more efficient for accessing elements by index.
* std::array: A fixed-size array.
* std::deque: A double-ended queue that allows insertion and deletion at both ends. It is more flexible than std::list in terms of memory allocation.
* std::forward\_list: A singly-linked list that only allows forward traversal, which saves memory but lacks the backward traversal feature.

## Pros and Cons

### Pros

* **Efficient Insertions/Deletions**: Inserting or deleting elements at any position (beginning, middle, or end) is O(1) since it only involves changing pointers.
* **Dynamic Size**: The size of the list can grow or shrink dynamically, accommodating an arbitrary number of elements without pre-allocation. Also, memory is allocated only as needed for new elements, reducing waste compared to pre-allocated structures.

### Cons

* **Memory Overhead**: Each element requires **additional memory for storing pointers** (to the previous and next nodes), leading to higher memory usage than arrays.
* **No Random Access**: Elements **cannot be accessed via an index**, making retrieval of elements slower (O(n)) compared to std::array or std::vector (O(1)).
* **Cache Locality**: Since elements are stored non-contiguously, cache performance may suffer, leading to slower access times compared to std::array or std::vector.

## Iteration

### Traditional For Loop with Iterators

#include <iostream>

#include <list>

int main() {

std::list<int> lst = {1, 2, 3, 4, 5};

for (std::list<int>::iterator it = lst.begin(); it != lst.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Range-Based For Loop

#include <iostream>

#include <list>

int main() {

std::list<int> lst = {1, 2, 3, 4, 5};

for (const auto& value : lst) {

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

}

### Using Const Iterators

#include <iostream>

#include <list>

int main() {

std::list<int> lst = {1, 2, 3, 4, 5};

for (std::list<int>::const\_iterator it = lst.cbegin(); it != lst.cend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Using std::for\_each

#include <iostream>

#include <list>

#include <algorithm>

int main() {

std::list<int> lst = {1, 2, 3, 4, 5};

std::for\_each(lst.begin(), lst.end(), [](int value) {

std::cout << value << " ";

});

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| push\_back | const T& value | Adds an element to the end of the list. | O(1) | O(1) | lst.push\_back(10); | cplusplus |
| push\_front | const T& value | Adds an element to the front of the list. | O(1) | O(1) | lst.push\_front(5); | cplusplus |
| pop\_back | None | Removes the last element from the list. | O(1) | O(1) | lst.pop\_back(); | cplusplus |
| pop\_front | None | Removes the first element from the list. | O(1) | O(1) | lst.pop\_front(); | cplusplus |
| insert | iterator pos,  const T& value | Inserts an element before the specified position. | O(1) | O(1) | lst.insert(  lst.begin(), 0); | cplusplus |
| erase | iterator pos | Removes the element at the specified position. | O(1) | O(1) | lst.erase(lst.begin()); | cplusplus |
| clear | None | Removes all elements from the list. | O(n) | O(1) | lst.clear(); | cplusplus |
| size | None | Returns the number of elements in the list. | O(1) | O(1) | size\_t s = lst.size(); | cplusplus |
| front | None | Returns a reference to the first element. | O(1) | O(1) | int first = lst.front(); | cplusplus |
| back | None | Returns a reference to the last element. | O(1) | O(1) | int last = lst.back(); | cplusplus |
| sort | None | Sorts the elements in the list. | O(nlog n) | O(1) | lst.sort(); | cplusplus |
| reverse | None | Reverses the order of elements in the list. | O(n) | O(1) | lst.reverse(); | cplusplus |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <list>  #include <algorithm>    int main() {      // Create and initialize a std::list      std::list<int> lst = {3, 1, 4, 1, 5};        // Add elements      lst.push\_back(9);      lst.push\_front(2);        // Display elements      std::cout << "List after adding elements: ";      for (const auto& value : lst) {          std::cout << value << " ";      }      std::cout << std::endl;        // Remove elements      lst.pop\_back();      lst.pop\_front();      std::cout << "List after popping elements: ";      for (const auto& value : lst) {          std::cout << value << " ";      }      std::cout << std::endl;        // Insert an element      lst.insert(std::next(lst.begin(), 1), 6);      std::cout << "List after inserting 6: ";      for (const auto& value : lst) {          std::cout << value << " ";      }      std::cout << std::endl;        // Sort the list      lst.sort();      std::cout << "Sorted list in ascending: ";      for (const auto& value : lst) {          std::cout << value << " ";      }      std::cout << std::endl;        // Sort the list in descending order      lst.sort([](int a, int b) {          return a > b;      });      std::cout << "Sorted list in descending order: ";      for (const auto& value : lst) {          std::cout << value << " ";      }      std::cout << std::endl;        // Clear the list      lst.clear();      std::cout << "List size after clear: " << lst.size() << std::endl;        return 0;  } | List after adding elements: 2 3 1 4 1 5 9  List after popping elements: 3 1 4 1 5  List after inserting 6: 3 6 1 4 1 5  Sorted list in ascending: 1 1 3 4 5 6  Sorted list in descending order: 6 5 4 3 1 1  List size after clear: 0 |

# Set

Set dynamically allocates memory for its elements. Each element is stored in a node of the tree, which holds pointers to its parent and children.

Values in a set are immutable. They cannot be modified once added to the set, though it’s possible to remove and then add the modified values of that element.

## What Is std::set?

std::set is an associative container that stores **unique** elements following a **specific order**. It is implemented as a *balanced binary search tree* (typically a *Red-Black Tree*), which allows for efficient retrieval, insertion, and deletion of elements.

### Pros

* **Automatic sorting**: Elements are stored in sorted order based on a comparison predicate (default is std::less<T>). This allows for efficient searching and insertion.
* **Efficient operations**: Operations like insertion, deletion, and lookup are O(logn) due to the underlying balanced tree structure.

### Cons

* **Memory overhead**: Each element requires extra memory for storing pointers and tree structure metadata, leading to higher memory usage compared to arrays.
* **No random access**: Elements cannot be accessed by index, which may limit use cases where random access is required.
* **Order maintenance**: Operations are slower than in hash-based containers like std::unordered\_set, which provide average O(1) complexity.

## What Is std::unordered\_set?

std::unordered\_set is the **unordered** version of std::set, which uses a *hash table* for faster insertions and searches, but does not maintain order.

### Pros

* **No order maintenance**: Does not require maintaining order, which can lead to better performance in scenarios where order is not needed.
* **Fast operations**: With hast table, operations like insertion, deletion, and lookup are average O(1), making it very suitable for large datasets.

However, in the worst case (e.g., many collisions), it can degrade to O(n), but this scenario is rare with a good hash function and proper load factor.

### Cons

* **Memory overhead**: Each element requires extra memory for storing pointers and buckets, leading to higher memory usage compared to arrays.
* **No random access**: Elements cannot be accessed by index, which may limit use cases where random access is required.
* **No automatic sorting**: Elements are not stored in any particular order, which may complicate operations that require sorted data.
* **No defined order**: Elements are organized based on their hash values in a hash table.

When you insert an element, it is placed into a bucket determined by its hash value. The physical position of the element within the bucket is determined by the hash table's internal structure, which does not guarantee an order.

When you iterate, the **order of elements may appear random** and can change whenever elements are added or removed. This is due to the way hashing works and how the underlying data structure may resize or rehash.

## What Is std::multiset?

Similar to std::set, but allows **duplicate** elements.

## What Is std::unordered\_multiset?

Similar to std::unordered\_set, but allows **duplicate** elements.

## Iteration

The iteration for std::set, std::unordered\_set, std::multiset, std::unordered\_multiset is the same.

### Range-Based For Loop

#include <iostream>

#include <set>

int main() {

std::set<int> mySet = {5, 3, 8, 1, 4};

for (const auto& value : mySet) {

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

}

### Using Iterators

#include <iostream>

#include <set>

int main() {

std::set<int> mySet = {5, 3, 8, 1, 4};

for (std::set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Using Const Iterators

#include <iostream>

#include <set>

int main() {

std::set<int> mySet = {5, 3, 8, 1, 4};

for (std::set<int>::const\_iterator it = mySet.cbegin(); it != mySet.cend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Using std::for\_each

#include <iostream>

#include <set>

#include <algorithm>

int main() {

std::set<int> mySet = {5, 3, 8, 1, 4};

std::for\_each(mySet.begin(), mySet.end(), [](int value) {

std::cout << value << " ";

});

std::cout << std::endl;

return 0;

}

## Built-in Functions

### std::set

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference** |
| insert | const T& value | Inserts a new element into the set. | O(logn) | O(1) | mySet.insert(10); | [cplusplus](https://cplusplus.com/reference/set/set/insert/) |
| erase | const T& value or iterator pos | Removes the specified element or the element at the given position. | O(logn) | O(1) | mySet.erase(10); | [cplusplus](https://cplusplus.com/reference/set/set/erase/) |
| find | const T& value | Finds an element in the set, returning an iterator to it. | O(logn) | O(1) | auto it = mySet.find(10); | [cplusplus](https://cplusplus.com/reference/set/set/find/) |
| count | const T& value | Returns the number of elements matching the value (0 or 1). | O(logn) | O(1) | size\_t c = mySet.count(10); | [cplusplus](https://cplusplus.com/reference/set/set/count/) |
| clear | None | Removes all elements from the set. | O(n) | O(1) | mySet.clear(); | [cplusplus](https://cplusplus.com/reference/set/set/clear/) |
| size | None | Returns the number of elements in the set. | O(1) | O(1) | size\_t s = mySet.size(); | [cplusplus](https://cplusplus.com/reference/set/set/size/) |
| empty | None | Checks whether the set is empty. | O(1) | O(1) | bool isEmpty = mySet.empty(); | [cplusplus](https://cplusplus.com/reference/set/set/empty/) |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = mySet.begin(); | [cplusplus](https://cplusplus.com/reference/set/set/begin/) |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = mySet.end(); | [cplusplus](https://cplusplus.com/reference/set/set/end/) |
| lower\_bound | const T& value | Returns an iterator to the first element that is not less than the given value. | O(logn) | O(1) | auto it = mySet.lower\_bound(4); | [cplusplus](https://cplusplus.com/reference/set/set/lower_bound/) |
| upper\_bound | const T& value | Returns an iterator to the first element that is greater than the given value. | O(logn) | O(1) | auto it = mySet.upper\_bound(4); | [cplusplus](https://cplusplus.com/reference/set/set/upper_bound/) |

### set::unordered\_set

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference** |
| insert | const T& value | Inserts a new element into the unordered set. | O(1) | O(1) | mySet.insert(10); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/insert/) |
|  | initializer\_list<T> il | Inserts elements from an initializer list. | O(n) | O(n) | mySet.insert({1, 2, 3}); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/insert/) |
|  | iterator pos, const T& value | Inserts the element at the specified position. | O(1) | O(1) | mySet.insert(mySet.end(), 20); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/insert/) |
| erase | const T& value | Removes the specified element from the set. | O(1) | O(1) | mySet.erase(10); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/erase/) |
|  | iterator pos | Removes the element at the specified position. | O(1) | O(1) | mySet.erase(mySet.begin()); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/erase/) |
|  | iterator first, iterator last | Removes the elements in the range [first, last). | O(n) | O(1) | mySet.erase(mySet.begin(), mySet.end()); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/erase/) |
| find | const T& value | Finds an element in the set, returning an iterator to it. | O(1) | O(1) | auto it = mySet.find(10); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/find/) |
| count | const T& value | Returns the number of elements matching the value (0 or 1). | O(1) | O(1) | size\_t c = mySet.count(10); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/count/) |
| clear | None | Removes all elements from the set. | O(n) | O(1) | mySet.clear(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/clear/) |
| size | None | Returns the number of elements in the set. | O(1) | O(1) | size\_t s = mySet.size(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/size/) |
| empty | None | Checks whether the set is empty. | O(1) | O(1) | bool isEmpty = mySet.empty(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/empty/) |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = mySet.begin(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/begin/) |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = mySet.end(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/end/) |
| bucket\_count | None | Returns the number of buckets used by the unordered set. | O(1) | O(1) | size\_t bc = mySet.bucket\_count(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/bucket_count/) |
| load\_factor | None | Returns the current load factor. | O(1) | O(1) | double lf = mySet.load\_factor(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/load_factor/) |
| max\_load\_factor | None | Returns the maximum load factor. | O(1) | O(1) | double ml = mySet.max\_load\_factor(); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/max_load_factor/) |
| rehash | size\_type count | Resizes the container to contain at least count buckets. | O(n) | O(n) | mySet.rehash(20); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/rehash/) |
| reserve | size\_type count | Requests that the capacity be at least enough to contain count elements. | O(n) | O(n) | mySet.reserve(50); | [cplusplus](http://www.cplusplus.com/reference/unordered_set/unordered_set/reserve/) |

### std::multiset

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference** |
| insert | const T& value | Inserts a new element into the multiset. | O(logn) | O(1) | mySet.insert(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/insert) |
|  | initializer\_list<T> il | Inserts elements from an initializer list. | O(n) | O(n) | mySet.insert({1, 2, 3}); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/insert) |
|  | iterator pos,  const T& value | Inserts the element at the specified position. | O(logn) | O(1) | mySet.insert(mySet.begin(), 20); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/insert) |
| erase | const T& value | Removes all elements matching the specified value. | O(logn) | O(1) | mySet.erase(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/erase) |
|  | iterator pos | Removes the element at the specified position. | O(logn) | O(1) | mySet.erase(mySet.begin()); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/erase) |
|  | iterator first,  iterator last | Removes the elements in the range [first, last). | O(n) | O(1) | mySet.erase(mySet.begin(), mySet.end()); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/erase) |
| find | const T& value | Finds an element in the multiset, returning an iterator. | O(logn) | O(1) | auto it = mySet.find(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/find) |
| count | const T& value | Returns the number of occurrences of the specified value. | O(logn) | O(1) | size\_t c = mySet.count(2); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/count) |
| clear | None | Removes all elements from the multiset. | O(n) | O(1) | mySet.clear(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/clear) |
| size | None | Returns the number of elements in the multiset. | O(1) | O(1) | size\_t s = mySet.size(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/size) |
| empty | None | Checks whether the multiset is empty. | O(1) | O(1) | bool isEmpty = mySet.empty(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/empty) |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = mySet.begin(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/begin) |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = mySet.end(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/end) |
| rbegin | None | Returns a reverse iterator to the last element. | O(1) | O(1) | auto rit = mySet.rbegin(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/rbegin) |
| rend | None | Returns a reverse iterator to the element before the first. | O(1) | O(1) | auto rit = mySet.rend(); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/rend) |
| equal\_range | const T& value | Returns a range of elements matching the specified value. | O(logn) | O(1) | auto range = mySet.equal\_range(2); | [cplusplus](https://en.cppreference.com/w/cpp/container/multiset/equal_range) |

### std::unordered\_multiset

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference** |
| insert | const T& value | Inserts a new element into the unordered multiset. | O(1) | O(1) | mySet.insert(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/insert) |
|  | initializer\_list<T> il | Inserts elements from an initializer list. | O(n) | O(n) | mySet.insert({1, 2, 3}); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/insert) |
|  | iterator pos,  const T& value | Inserts the element at the specified position. | O(1) | O(1) | mySet.insert(mySet.end(), 20); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/insert) |
| erase | const T& value | Removes all elements matching the specified value. | O(1) | O(1) | mySet.erase(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/erase) |
|  | iterator pos | Removes the element at the specified position. | O(1) | O(1) | mySet.erase(mySet.begin()); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/erase) |
|  | iterator first,  iterator last | Removes the elements in the range [first, last). | O(n) | O(1) | mySet.erase(mySet.begin(), mySet.end()); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/erase) |
| find | const T& value | Finds an element in the unordered multiset, returning an iterator. | O(1) | O(1) | auto it = mySet.find(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/find) |
| count | const T& value | Returns the number of occurrences of the specified value. | O(1) | O(1) | size\_t c = mySet.count(2); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/count) |
| clear | None | Removes all elements from the unordered multiset. | O(n) | O(1) | mySet.clear(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/clear) |
| size | None | Returns the number of elements in the unordered multiset. | O(1) | O(1) | size\_t s = mySet.size(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/size) |
| empty | None | Checks whether the unordered multiset is empty. | O(1) | O(1) | bool isEmpty = mySet.empty(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/empty) |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = mySet.begin(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/begin) |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = mySet.end(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/end) |
| bucket\_count | None | Returns the number of buckets used by the unordered multiset. | O(1) | O(1) | size\_t bc = mySet.bucket\_count(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/bucket_count) |
| load\_factor | None | Returns the current load factor. | O(1) | O(1) | double lf = mySet.load\_factor(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/load_factor) |
| max\_load\_factor | None | Returns the maximum load factor. | O(1) | O(1) | double ml = mySet.max\_load\_factor(); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/max_load_factor) |
| rehash | size\_type count | Resizes the container to contain at least count buckets. | O(n) | O(n) | mySet.rehash(20); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/rehash) |
| reserve | size\_type count | Requests that the capacity be at least enough to contain count elements. | O(n) | O(n) | mySet.reserve(50); | [cplusplus](https://en.cppreference.com/w/cpp/container/unordered_multiset/reserve) |

## Example

### std::set

|  |  |
| --- | --- |
| #include <iostream>  #include <set>  #include <algorithm>    int main() {  // Create and initialize a std::set  std::set<int> mySet = {5, 3, 8, 1, 1, 4};    // Insert an element  mySet.insert(10);  std::cout << "Set after insertion: ";  for (const auto& value : mySet) {  std::cout << value << " ";  }  std::cout << std::endl;    // Remove an element  mySet.erase(3);  std::cout << "Set after erasing 3: ";  for (const auto& value : mySet) {  std::cout << value << " ";  }  std::cout << std::endl;    // Check if an element exists  auto it = mySet.find(8);  if (it != mySet.end()) {  std::cout << "Found 8 in the set." << std::endl;  } else {  std::cout << "8 not found in the set." << std::endl;  }    // Count elements  size\_t count = mySet.count(10);  std::cout << "Count of 10 in the set: " << count << std::endl;    // Display size and check if empty  std::cout << "Size of the set: " << mySet.size() << std::endl;  std::cout << "Is the set empty? " << (mySet.empty() ? "Yes" : "No") << std::endl;    // Lower and upper bounds  auto lower = mySet.lower\_bound(4);  auto upper = mySet.upper\_bound(4);  std::cout << "Lower bound of 4: " << (lower != mySet.end() ? \*lower : -1) << std::endl;  std::cout << "Upper bound of 4: " << (upper != mySet.end() ? \*upper : -1) << std::endl;      // Create another set called anotherSet, then assign the elements from mySet to anotherSet  std::set<int> anotherSet(mySet.begin(), mySet.end());  std::cout << "New set is: ";  for (auto it = anotherSet.begin(); it != anotherSet.end(); it++) {  std::cout << \*it << " ";  }  std::cout << std::endl;    // Remove all elements less than 5 in set  std::cout << "Set after removal of elements less than 5: ";  mySet.erase(mySet.begin(), mySet.find(5));  for (it = mySet.begin(); it != mySet.end(); it++) {  std::cout << \*it << " ";  }  std::cout << std::endl;    // Clear the set  mySet.clear();  std::cout << "Size of the set after clearing: " << mySet.size() << std::endl;    return 0;  } | Set after insertion: 1 3 4 5 8 10  Set after erasing 3: 1 4 5 8 10  Found 8 in the set.  Count of 10 in the set: 1  Size of the set: 5  Is the set empty? No  Lower bound of 4: 4  Upper bound of 4: 5  The new set is: 1 4 5 8 10  The set after removal of elements less than 5: 5 8 10  Size of the set after clearing: 0 |

### std::unordered\_set

|  |  |
| --- | --- |
| #include <iostream>  #include <unordered\_set>  #include <algorithm>    int main() {      // Create and initialize a std::unordered\_set      std::unordered\_set<int> mySet = {5, 3, 8, 1, 1, 4};        // Insert an element      mySet.insert(10);      std::cout << "Set after insertion: ";      for (const auto& value : mySet) {          std::cout << value << " ";      }      std::cout << std::endl;        // Remove an element      mySet.erase(3);      std::cout << "Set after erasing 3: ";      for (const auto& value : mySet) {          std::cout << value << " ";      }      std::cout << std::endl;        // Check if an element exists      auto it = mySet.find(8);      if (it != mySet.end()) {          std::cout << "Found 8 in the set." << std::endl;      } else {          std::cout << "8 not found in the set." << std::endl;      }        // Count elements      size\_t count = mySet.count(10);      std::cout << "Count of 10 in the set: " << count << std::endl;        // Display size and check if empty      std::cout << "Size of the set: " << mySet.size() << std::endl;      std::cout << "Is the set empty? " << (mySet.empty() ? "Yes" : "No") << std::endl;        // Create another set called anotherSet, then assign the elements from mySet to anotherSet      std::unordered\_set<int> anotherSet(mySet.begin(), mySet.end());      std::cout << "New set is: ";      for (auto it = anotherSet.begin(); it != anotherSet.end(); it++) {          std::cout << \*it << " ";      }      std::cout << std::endl;        // Clear the set      mySet.clear();      std::cout << "Size of the set after clearing: " << mySet.size() << std::endl;        return 0;  } | Set after insertion: 10 4 1 8 3 5  Set after erasing 3: 10 4 1 8 5  Found 8 in the set.  Count of 10 in the set: 1  Size of the set: 5  Is the set empty? No  New set is: 5 8 1 4 10  Size of the set after clearing: 0 |

## Advances

### Sets of User-Defined Types

**1.** If ordered set stores user-defined data type (class or struct), have to overload operator **<** so the set knows how to sort its items.

Other ways: <https://stackoverflow.com/a/46128321/14835442>

**2.** There’re some ways to find item from set:

1. set::find(): It uses overload operator <. So if set stores user-defined data type, have to overload operator <.
2. std::find(): It uses overload operator ==. So if set stores user-defined data type, have to overload operator ==.
3. std::find\_if(): It doesn’t uses overload operator. So we don’t have to overload any operator, but have to write a comparision function to pass to find\_if().
4. Manually iterate each item and write your own code.

**3.** Operator overloads work with **objects, not pointers**. So if your set is std::set<object\*>, it cannot work with operator overload.

The solution is to create custom comparator function or functor, and pass it to the set. This is similar to "[Other ways](https://stackoverflow.com/a/46128321/14835442)" in note 1 (just replace object with pointer).

### Careful When Erasing

**1**. Don't increase the iterator when ERASING element:

|  |  |
| --- | --- |
| #include <iostream>  #include <set>  using namespace std;    int main() {  set<int> s{ 1, 2, 3, 4, 5 };  set<int>::iterator it;    // BAD WAY  // In Visual Studio, this code will crash  // with exception "Expression: cannot increment value-initialized map/set iterator"  // because we tried to increase "it" even when it is "end"  int i = 0;  for (it = s.begin(); it != s.end(); ++it) {  if (i == 4) {  it = s.erase(it); // After erase, "it" will be increase by 1  // So when "it" is "end", "++it" in the loop will cause exception  }  i++;  }    // GOOD WAY  // Advice here: If you erase, you should not increase the iterator (more [details](https://stackoverflow.com/questions/72869355/stdset-using-iterator-causes-memory-violation-exception))  int i = 0;  for (it = s.begin(); it != s.end(); ) {  if (i == 4) {  it = s.erase(it); // After erase, "it" will be increase by 1  }  else {  it++; // Increase "it" by 1  }  i++;  }    // Print all elements in the set  for (auto x : ms) {  cout << x << " ";  }    return 0;  } | 1 2 3 4 |

# Tuple

[Tuples in C++ - GeeksforGeeks](https://www.geeksforgeeks.org/tuples-in-c/)

[tie - C++ Reference (cplusplus.com)](https://cplusplus.com/reference/tuple/tie/)

# Pair

## What Is std::pair?

### Definition

std::pair is a simple data structure that allows you to **store two heterogeneous values as a single unit**.

### Memory Management

* **Dynamic and Static**: Pair manages its memory statically. When you create a pair, the memory for both of its elements is allocated on the stack if it's a local variable or in the heap if it is dynamically allocated.

### Similar Data Structures

* std::tuple: Similar to std::pair, but can hold more than two values of potentially different types.

## Pros and Cons

### Pros

* **Simplicity**: Easy to use for grouping two related values without defining a separate struct or class.
* **Heterogeneous types**: Can store two values of different types, which is useful for returning multiple values from functions.
* **Lightweight**: Minimal overhead compared to more complex data structures, making it efficient in terms of performance.

### Cons

* **Not suitable for complex data**: Only stores two values; for more, you need to use std::tuple or other structures.
* **No named members**: Accessing values is done via first and second, which may not be as descriptive as named struct or class members.

## Access

### Accessing Elements Directly

#include <iostream>

#include <utility>

int main() {

std::pair<int, std::string> myPair(1, "Hello");

std::cout << "First: " << myPair.first << std::endl;

std::cout << "Second: " << myPair.second << std::endl;

return 0;

}

### Using Structured Bindings (C++17)

#include <iostream>

#include <utility>

int main() {

std::pair<int, std::string> myPair(1, "Hello");

auto [first, second] = myPair;

std::cout << "First: " << first << ", Second: " << second << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| std::pair | const T1& first,  const T2& second | Constructs a pair with specified values. | O(1) | O(1) | std::pair<int, std::string> p(1, "Hello"); | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/pair) |
| std::make\_pair | const T1& first,  const T2& second | Constructs a pair with specified values, deducing types. | O(1) | O(1) | auto p = std::make\_pair(1, "Hello"); | [cplusplus](https://en.cppreference.com/w/cpp/utility/make_pair) |
| first | None | Accesses the first element of the pair. | O(1) | O(1) | p.first = 1; | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/first) |
| second | None | Accesses the second element of the pair. | O(1) | O(1) | p.second = "World"; | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/second) |
| operator= | const pair& other | Assigns the value of one pair to another. | O(1) | O(1) | p1 = p2; | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator=) |
| operator== | const pair& other | Compares two pairs for equality. | O(1) | O(1) | if (p1 == p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator==) |
| operator!= | const pair& other | Compares two pairs for inequality. | O(1) | O(1) | if (p1 != p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator!=) |
| operator< | const pair& other | Compares two pairs lexicographically. | O(1) | O(1) | if (p1 < p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator%3C) |
| operator> | const pair& other | Compares two pairs lexicographically. | O(1) | O(1) | if (p1 > p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator%3E) |
| operator<= | const pair& other | Compares two pairs lexicographically. | O(1) | O(1) | if (p1 <= p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator%3C=) |
| operator>= | const pair& other | Compares two pairs lexicographically. | O(1) | O(1) | if (p1 >= p2) { /\* ... \*/ } | [cplusplus](https://en.cppreference.com/w/cpp/utility/pair/operator%3E=) |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <utility>  #include <vector>  #include <algorithm>    int main() {  // Create a vector of pairs  std::vector<std::pair<int, std::string>> vec;    // Insert pairs into the vector  vec.emplace\_back(1, "Apple");  vec.emplace\_back(3, "Banana");  vec.emplace\_back(2, "Cherry");    // Display the pairs  std::cout << "Pairs in the vector:" << std::endl;  for (const auto& p : vec) {  std::cout << "First: " << p.first << ", Second: " << p.second << std::endl;  }    // Sort the vector of pairs based on the first element  std::sort(vec.begin(), vec.end());    // Display sorted pairs  std::cout << "\nSorted pairs:" << std::endl;  for (const auto& p : vec) {  std::cout << "First: " << p.first << ", Second: " << p.second << std::endl;  }    // Compare pairs  std::pair<int, std::string> p1(1, "Apple");  std::pair<int, std::string> p2(1, "Apple");    if (p1 == p2) {  std::cout << "\nThe pairs p1 and p2 are equal." << std::endl;  }    return 0;  } | Pairs in the vector:  First: 1, Second: Apple  First: 3, Second: Banana  First: 2, Second: Cherry  Sorted pairs:  First: 1, Second: Apple  First: 2, Second: Cherry  First: 3, Second: Banana  The pairs p1 and p2 are equal |

# Map

## What Is std::map?

An associative container that stores elements in **key-value pairs** following a **specific order**, where the **keys are unique**. It is implemented as a *balanced binary tree* (usually a *Red-Black tree*), which allows for O(logn) time complexity for search, insertion, and deletion.

### Pros

* **Logarithmic access time**: Efficient insertion, deletion, and lookup.

### Cons

* **Overhead**: More memory and time overhead compared to unordered maps due to tree structures.
* **Slower access**: Generally slower than std::unordered\_map for access operations.

## What Is std::unordered\_map?

An associative container that stores elements in key-value pairs like std::map, but without **any specific order**. It uses a *hash table* for storage, which provides average O(1) time complexity for search, insertion, and deletion.

### Pros:

* **Fast access**: Average constant time performance for search, insert, and delete.

### Cons

* **Potential collisions**: Performance can degrade if there are many hash collisions.

## What Is std::multimap?

Similar to std::map, but allows **multiple values for the same key**. This means that keys are not unique.

## What Is std::unordered\_multimap?

Similar to std::unordered\_map, but allows **multiple values for the same key**, storing them in a hash table.

## Iteration

### Using Range-Based For Loop

#include <iostream>

#include <map>

int main() {

std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}, {3, "Three"}};

for (const auto& pair : myMap) {

std::cout << pair.first << ": " << pair.second << std::endl;

}

return 0;

}

### Using Iterators

#include <iostream>

#include <map>

int main() {

std::map<int, std::string> myMap = {{1, "Apple"}, {2, "Banana"}, {3, "Cherry"}};

for (auto it = myMap.begin(); it != myMap.end(); ++it) {

std::cout << it->first << ": " << it->second << std::endl;

}

return 0;

}

### Using std::for\_each with a Lambda Function

#include <iostream>

#include <map>

#include <algorithm>

int main() {

std::map<int, std::string> myMap = {{1, "Dog"}, {2, "Cat"}, {3, "Bird"}};

std::for\_each(myMap.begin(), myMap.end(), [](const std::pair<int, std::string>& pair) {

std::cout << pair.first << ": " << pair.second << std::endl;

});

return 0;

}

## Built-in Functions

### std::map

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| insert | const value\_type& val | Inserts a new element. | O(logn) | O(1) | myMap.insert({1, "Hello"}); | [cplusplus](https://cplusplus.com/reference/map/map/insert/) |
|  | iterator hint,  const value\_type& val | Inserts a new element with a hint. | O(logn) | O(1) | myMap.insert(myMap.end(), {2, "World"}); | cplusplus |
| erase | const key\_type& key | Removes element by key. | O(logn) | O(1) | myMap.erase(1); | cplusplus |
|  | iterator pos | Removes element at position. | O(1) | O(1) | myMap.erase(myMap.begin()); | cplusplus |
| find | const key\_type& key | Finds an element by key. | O(logn) | O(1) | auto it = myMap.find(2); | cplusplus |
| clear | None | Removes all elements. | O(n) | O(1) | myMap.clear(); | cplusplus |
| size | None | Returns the number of elements. | O(1) | O(1) | size\_t s = myMap.size(); | cplusplus |
| empty | None | Checks if the map is empty. | O(1) | O(1) | bool isEmpty = myMap.empty(); | cplusplus |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = myMap.begin(); | cplusplus |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = myMap.end(); | cplusplus |

### std::unordered\_map

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| insert | const value\_type& val | Inserts a new element. | O(1) | O(1) | myMap.insert({1, "Hello"}); | cplusplus |
|  | iterator hint,  const value\_type& val | Inserts a new element with a hint. | O(1) | O(1) | myMap.insert(myMap.end(), {2, "World"}); | cplusplus |
| erase | const key\_type& key | Removes element by key. | O(1) | O(1) | myMap.erase(1); | cplusplus |
|  | iterator pos | Removes element at position. | O(1) | O(1) | myMap.erase(myMap.begin()); | cplusplus |
| find | const key\_type& key | Finds an element by key. | O(1) | O(1) | auto it = myMap.find(2); | cplusplus |
| clear | None | Removes all elements. | O(n) | O(1) | myMap.clear(); | cplusplus |
| size | None | Returns the number of elements. | O(1) | O(1) | size\_t s = myMap.size(); | cplusplus |
| empty | None | Checks if the map is empty. | O(1) | O(1) | bool isEmpty = myMap.empty(); | cplusplus |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = myMap.begin(); | cplusplus |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = myMap.end(); | cplusplus |

### std::multimap

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| insert | const value\_type& val | Inserts a new element. | O(logn) | O(1) | myMap.insert({1, "Hello"}); | cplusplus |
| erase | const key\_type& key | Removes all elements with the specified key. | O(logn) | O(1) | myMap.erase(1); | cplusplus |
|  | iterator pos | Removes element at position. | O(1) | O(1) | myMap.erase(myMap.begin()); | cplusplus |
| find | const key\_type& key | Finds an element by key. | O(logn) | O(1) | auto it = myMap.find(2); | cplusplus |
| clear | None | Removes all elements. | O(n) | O(1) | myMap.clear(); | cplusplus |
| size | None | Returns the number of elements. | O(1) | O(1) | size\_t s = myMap.size(); | cplusplus |
| empty | None | Checks if the map is empty. | O(1) | O(1) | bool isEmpty = myMap.empty(); | cplusplus |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = myMap.begin(); | cplusplus |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = myMap.end(); | cplusplus |

### std::unordered\_multimap

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| insert | const value\_type& val | Inserts a new element. | O(1) | O(1) | myMap.insert({1, "Hello"}); | cplusplus |
| erase | const key\_type& key | Removes all elements with the specified key. | O(1) | O(1) | myMap.erase(1); | cplusplus |
|  | iterator pos | Removes element at position. | O(1) | O(1) | myMap.erase(myMap.begin()); | cplusplus |
| find | const key\_type& key | Finds an element by key. | O(1) | O(1) | auto it = myMap.find(2); | cplusplus |
| clear | None | Removes all elements. | O(n) | O(1) | myMap.clear(); | cplusplus |
| size | None | Returns the number of elements. | O(1) | O(1) | size\_t s = myMap.size(); | cplusplus |
| empty | None | Checks if the map is empty. | O(1) | O(1) | bool isEmpty = myMap.empty(); | cplusplus |
| begin | None | Returns an iterator to the first element. | O(1) | O(1) | auto it = myMap.begin(); | cplusplus |
| end | None | Returns an iterator to the past-the-end element. | O(1) | O(1) | auto it = myMap.end(); | cplusplus |

## Example

### std::map

|  |  |
| --- | --- |
| #include <iostream>  #include <map>    int main() {      // Create an ordered map      std::map<int, int> m1;      m1[1] = 40;      m1[2] = 30;      m1[3] = 60;      m1[4] = 20;  // Another way of inserting a value in a map      m1.insert(std::pair<int, int>(7, 10));      m1.insert(std::pair<int, int>(5, 50));      m1.insert(std::pair<int, int>(6, 50));  m1.insert(std::pair<int, int>(6, 90));        // Print all elements in map m1      std::map<int, int>::iterator it;      std::cout << "\nMap m1 is:\n";      std::cout << "\tKEY\tVALUE\n";      for (it = m1.begin(); it != m1.end(); ++it) {          std::cout << '\t' << it->first << '\t' << it->second << std::endl;      }        // Create another map called m2, then assign all elements from m1 to m2      std::map<int, int> m2(m1.begin(), m1.end());        // Remove all elements with keys smaller than 3 in m2      std::cout << "\nMap m2 after removal of elements with keys < 3:\n";      std::cout << "\tKEY\tVALUE\n";      m2.erase(m2.begin(), m2.find(3));      for (it = m2.begin(); it != m2.end(); ++it) {          std::cout << '\t' << it->first << '\t' << it->second << std::endl;      }        // Remove an element with key 4 in m2      int num = m2.erase(4);      std::cout << "\nMap m2 after removal of element at key 4:\n";      std::cout << "  ==> " << num << " element is removed\n";      std::cout << "\tKEY\tVALUE\n";      for (it = m2.begin(); it != m2.end(); ++it) {          std::cout << '\t' << it->first << '\t' << it->second << std::endl;      }        return 0;  } | Map m1 is:  KEY VALUE  1 40  2 30  3 60  4 20  5 50  6 50  7 10  Map m2 after removal of elements with keys < 3:  KEY VALUE  3 60  4 20  5 50  6 50  7 10  Map m2 after removal of element at key 4:  ==> 1 element is removed  KEY VALUE  3 60  5 50  6 50  7 10 |

### std::unordered\_map

|  |  |
| --- | --- |
| #include <iostream>  #include <map>    int main() {  // Create an ordered map  std::map<int, int> m1;  m1[1] = 40;  m1[2] = 30;  m1[3] = 60;  m1[4] = 20;    // Another way of inserting a value in a map  m1.insert(std::pair<int, int>(7, 10));  m1.insert(std::pair<int, int>(5, 50));  m1.insert(std::pair<int, int>(6, 50));  m1.insert(std::pair<int, int>(6, 90));    // Print all elements in map m1  std::map<int, int>::iterator it;  std::cout << "\nMap m1 is:\n";  std::cout << "\tKEY\tVALUE\n";  for (it = m1.begin(); it != m1.end(); ++it) {  std::cout << '\t' << it->first << '\t' << it->second << std::endl;  }    // Create another map called m2, then assign all elements from m1 to m2  std::map<int, int> m2(m1.begin(), m1.end());    // Remove all elements with keys smaller than 3 in m2  std::cout << "\nMap m2 after removal of elements with keys < 3:\n";  std::cout << "\tKEY\tVALUE\n";  m2.erase(m2.begin(), m2.find(3));  for (it = m2.begin(); it != m2.end(); ++it) {  std::cout << '\t' << it->first << '\t' << it->second << std::endl;  }    // Remove an element with key 4 in m2  int num = m2.erase(4);  std::cout << "\nMap m2 after removal of element at key 4:\n";  std::cout << " ==> " << num << " element is removed\n";  std::cout << "\tKEY\tVALUE\n";  for (it = m2.begin(); it != m2.end(); ++it) {  std::cout << '\t' << it->first << '\t' << it->second << std::endl;  }    return 0;  } | Map m1 is:  KEY VALUE  6 50  5 50  7 10  4 20  3 60  2 30  1 40  Map m2 after removal of element at key 4:  ==> 1 element is removed  KEY VALUE  1 40  2 30  3 60  7 10  5 50  6 50 |

## Advances

### Maps of User-Defined Types

**1.** If ordered map has key of user-defined data type (class or struct), have to overload operator **<** so the map knows how to sort its items.

Other ways: <https://stackoverflow.com/a/46128321/14835442> (samples are about set, but can apply similarly to map)

**2.** There’re some ways to find item from map:

1. map::find(): It uses overload operator <. So if set stores user-defined data type, have to overload operator <.
2. std::find\_if(): It doesn’t uses overload operator. So we don’t have to overload any operator, but have to write a comparision function to pass to find\_if().
3. Write your own searching function.

Note: std::find() doesn’t work with map.

**3.** Operator overloads work with **objects, not pointers**. So if your map is "std::map<object1\*, object2>", then it cannot work with operator overload.

The solution is to create custom comparator function or functor, and pass it to the map. This is similar to "[Other ways](https://stackoverflow.com/a/46128321/14835442)" in note 1 (just replace object with pointer).

### Sorting Maps

The std::sort() won’t work with map. So, to sort items in map, we can do one of following workaround ways:

1. Make another map, which uses the original map’s values as its keys and the original map’s keys as its values. A multimap is used because values of the original map can be duplicate.
2. Copy the map into a vector (or set, list, etc.) of key-value pairs. Then sort items in the vector using std::sort().
3. Write your own sorting function.

Examples [here](https://www.educative.io/answers/how-to-sort-a-map-by-value-in-cpp).

Note: STL map doesn’t provide map::sort().

# Stack

## What Is std::stack?

### Definition

std::stack is a *container adapter* that provides a Last-In-First-Out (LIFO) data structure. In a stack, the last element added is the first one to be removed. It allows operations such as push, pop, and access to the top element.

### Memory Management

* **Dynamic memory**: Stack manages memory dynamically. It typically uses an underlying container, such as std::deque or std::vector, to store elements. The underlying container handles memory allocation as elements are added or removed.
* **Ownership**: Stack does not manage memory for its elements directly; it relies on the underlying container. When the stack is destroyed, the underlying container is also destroyed, freeing its memory.

## Pros and Cons

### Pros

* **Efficient**: Operations like push and pop are O(1), making it very efficient for scenarios where you need to manage data in reverse order.

### Cons

* **Limited access**: You can only access the top element directly; you cannot access elements in the middle or bottom of the stack.
* **No iteration**: The std::stack does not provide iterators, making it difficult to traverse all elements.
* **Less flexibility**: It is specifically designed for LIFO operations, limiting its use cases compared to more versatile containers.

## Iteration

std::stack does not support iteration directly. However, you can copy the elements to another container or use a temporary stack to display the elements.

### Using a Temporary Stack

#include <iostream>

#include <stack>

void printStack(std::stack<int> s) {

while (!s.empty()) {

std::cout << s.top() << " ";

s.pop();

}

std::cout << std::endl;

}

int main() {

std::stack<int> myStack;

myStack.push(1);

myStack.push(2);

myStack.push(3);

printStack(myStack); // Outputs: 3 2 1

return 0;

}

### Using a Vector to Copy Elements

#include <iostream>

#include <stack>

#include <vector>

int main() {

std::stack<int> myStack;

myStack.push(1);

myStack.push(2);

myStack.push(3);

// Copying elements to a vector for iteration

std::vector<int> vec;

while (!myStack.empty()) {

vec.push\_back(myStack.top());

myStack.pop();

}

// Iterating through the vector

std::cout << "Stack elements: ";

for (const int& elem : vec) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| push | const value\_type& val | Adds an element to the top of the stack. | O(1) | O(1) | myStack.push(10); | [cplusplus](https://en.cppreference.com/w/cpp/container/stack/push) |
| pop | None | Removes the top element from the stack. | O(1) | O(1) | myStack.pop(); | [cplusplus](https://en.cppreference.com/w/cpp/container/stack/pop) |
| top | None | Accesses the top element of the stack. | O(1) | O(1) | int top = myStack.top(); | [cplusplus](https://en.cppreference.com/w/cpp/container/stack/top) |
| empty | None | Checks whether the stack is empty. | O(1) | O(1) | bool isEmpty = myStack.empty(); | [cplusplus](https://en.cppreference.com/w/cpp/container/stack/empty) |
| size | None | Returns the number of elements in the stack. | O(1) | O(1) | size\_t size = myStack.size(); | [cplusplus](https://en.cppreference.com/w/cpp/container/stack/size) |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <stack>    int main() {      std::stack<int> myStack;        // Pushing elements onto the stack      myStack.push(5);      myStack.push(10);      myStack.push(15);      myStack.push(20);        std::cout << "Stack size: " << myStack.size() << std::endl;        // Accessing the top element      std::cout << "Top element: " << myStack.top() << std::endl;        // Popping elements from the stack      std::cout << "Popping elements: ";      while (!myStack.empty()) {          std::cout << myStack.top() << " ";          myStack.pop();      }      std::cout << std::endl;        std::cout << "Stack size after popping: " << myStack.size() << std::endl;        return 0;  } | Stack size: 4  Top element: 20  Popping elements: 20 15 10 5  Stack size after popping: 0 |

# Queue

## What Is std::queue?

### Definition

std::queue is a *container adapter* that provides a First-In-First-Out (FIFO) data structure. In a queue, the first element added is the first one to be removed. This structure is ideal for scenarios where order of processing is important, such as task scheduling and breadth-first search algorithms.

### Memory Management

* **Dynamic memory**: std::queue manages memory dynamically using an underlying container, typically std::deque or std::list. This means it can grow and shrink as needed, allocating memory on the heap for elements as they are added or removed.
* **Ownership**: The queue itself does not manage the memory of its elements directly; it relies on the underlying container to handle allocation and deallocation.

## Pros and Cons

### Pros

* **Efficient operations**: Operations like enqueue (push) and dequeue (pop) are O(1), making it efficient for managing sequences of data.

### Cons

* **Limited access**: You can only access the front and back elements directly; you cannot access elements in the middle.
* **No iteration**: The std::queue does not provide iterators, making traversal of all elements difficult.
* **Less flexibility**: It is specifically designed for FIFO operations, limiting its use cases compared to more versatile containers.

## Iteration

std::queue does not support direct iteration. However, you can copy the elements to another container or use a temporary queue to display the elements.

### Using a Temporary Queue

#include <iostream>

#include <queue>

void printQueue(std::queue<int> q) {

while (!q.empty()) {

std::cout << q.front() << " ";

q.pop();

}

std::cout << std::endl;

}

int main() {

std::queue<int> myQueue;

myQueue.push(1);

myQueue.push(2);

myQueue.push(3);

printQueue(myQueue); // Outputs: 1 2 3

return 0;

}

### Using a Vector to Copy Elements

#include <iostream>

#include <queue>

#include <vector>

int main() {

std::queue<int> myQueue;

myQueue.push(1);

myQueue.push(2);

myQueue.push(3);

// Copying elements to a vector for iteration

std::vector<int> vec;

while (!myQueue.empty()) {

vec.push\_back(myQueue.front());

myQueue.pop();

}

// Iterating through the vector

std::cout << "Queue elements: ";

for (const int& elem : vec) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| push | const value\_type& val | Adds an element to the back of the queue. | O(1) | O(1) | myQueue.push(10); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/push/) |
| pop | None | Removes the front element from the queue. | O(1) | O(1) | myQueue.pop(); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/pop/) |
| front | None | Accesses the front element of the queue. | O(1) | O(1) | int frontElem = myQueue.front(); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/front/) |
| back | None | Accesses the last element of the queue. | O(1) | O(1) | int backElem = myQueue.back(); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/back/) |
| empty | None | Checks whether the queue is empty. | O(1) | O(1) | bool isEmpty = myQueue.empty(); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/empty/) |
| size | None | Returns the number of elements in the queue. | O(1) | O(1) | size\_t size = myQueue.size(); | [cplusplus](https://www.cplusplus.com/reference/queue/queue/size/) |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <queue>    int main() {  std::queue<int> myQueue;    // Pushing elements onto the queue  myQueue.push(5);  myQueue.push(10);  myQueue.push(15);  myQueue.push(20);    std::cout << "Queue size: " << myQueue.size() << std::endl;    // Accessing the front and back elements  std::cout << "Front element: " << myQueue.front() << std::endl;  std::cout << "Back element: " << myQueue.back() << std::endl;    // Popping elements from the queue  std::cout << "Popping elements: ";  while (!myQueue.empty()) {  std::cout << myQueue.front() << " ";  myQueue.pop();  }  std::cout << std::endl;    std::cout << "Queue size after popping: " << myQueue.size() << std::endl;    return 0;  } | Queue size: 4  Front element: 5  Back element: 20  Popping elements: 5 10 15 20  Queue size after popping: 0 |

# Double-Ended Queue

## What Is std::deque?

### Definition

std::deque (double-ended queue) is a sequence container that allows fast insertion and deletion at both the beginning and the end. It provides random access to its elements and can grow in both directions, making it versatile for various applications.

### Memory Management

* **Dynamic Memory**: std::deque manages memory dynamically. It typically allocates memory in chunks, which allows it to efficiently handle insertions and deletions from both ends without reallocating the entire structure.
* **Ownership**: The deque itself is responsible for managing its own memory, and it frees the memory allocated for its elements when it is destroyed.

### Similar Data Structures

* std::list: A doubly linked list that allows efficient insertions and deletions but has slower access times compared to std::deque.

## Pros and Cons

### Pros

* **Dynamic Size**: It can grow and shrink dynamically, allowing for efficient memory usage.
* **Fast Insertions/Deletions**: Insertion and deletion operations at both ends are O(1), making it suitable for queue-like operations.
* **Random Access**: Supports random access through the subscript operator [], allowing you to access elements efficiently.

### Cons

* **Memory Overhead**: It may have higher memory overhead compared to std::vector due to its internal structure and allocation strategy.
* **Cache Performance**: Due to its non-contiguous memory allocation, it may have poorer cache performance compared to std::vector.
* **Complexity**: Slightly more complex than simple arrays or vectors, which might not be needed for simpler use cases.

## Iteration

### Using a Range-Based For Loop

#include <iostream>

#include <deque>

int main() {

std::deque<int> myDeque = {1, 2, 3, 4, 5};

std::cout << "Elements in deque: ";

for (const auto& elem : myDeque) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

### Using Iterators

#include <iostream>

#include <deque>

int main() {

std::deque<int> myDeque = {1, 2, 3, 4, 5};

std::cout << "Elements in deque: ";

for (auto it = myDeque.begin(); it != myDeque.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

### Using Reverse Iterators

#include <iostream>

#include <deque>

int main() {

std::deque<int> myDeque = {1, 2, 3, 4, 5};

std::cout << "Elements in deque in reverse: ";

for (auto it = myDeque.rbegin(); it != myDeque.rend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| push\_back | const value\_type& val | Adds an element to the end of the deque. | O(1) | O(1) | myDeque.push\_back(10); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/push_back/) |
| push\_front | const value\_type& val | Adds an element to the front of the deque. | O(1) | O(1) | myDeque.push\_front(5); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/push_front/) |
| pop\_back | None | Removes the last element from the deque. | O(1) | O(1) | myDeque.pop\_back(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/pop_back/) |
| pop\_front | None | Removes the first element from the deque. | O(1) | O(1) | myDeque.pop\_front(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/pop_front/) |
| front | None | Accesses the first element of the deque. | O(1) | O(1) | int frontElem = myDeque.front(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/front/) |
| back | None | Accesses the last element of the deque. | O(1) | O(1) | int backElem = myDeque.back(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/back/) |
| empty | None | Checks whether the deque is empty. | O(1) | O(1) | bool isEmpty = myDeque.empty(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/empty/) |
| size | None | Returns the number of elements in the deque. | O(1) | O(1) | size\_t size = myDeque.size(); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/size/) |
| operator[] | size\_type pos | Accesses the element at the specified position. | O(1) | O(1) | int elem = myDeque[1]; | [cplusplus](https://www.cplusplus.com/reference/deque/deque/operator%5B%5D/) |
| at | size\_type pos | Accesses the element at the specified position with bounds checking. | O(1) | O(1) | int elem = myDeque.at(1); | [cplusplus](https://www.cplusplus.com/reference/deque/deque/at/) |

## Example

|  |  |
| --- | --- |
| #include <iostream>  #include <deque>    int main() {  std::deque<int> myDeque;    // Adding elements  myDeque.push\_back(10);  myDeque.push\_front(5);  myDeque.push\_back(15);  myDeque.push\_front(1);    std::cout << "Deque elements: ";  for (const auto& elem : myDeque) {  std::cout << elem << " ";  }  std::cout << std::endl;    // Accessing front and back elements  std::cout << "Front element: " << myDeque.front() << std::endl;  std::cout << "Back element: " << myDeque.back() << std::endl;    // Removing elements  myDeque.pop\_front();  myDeque.pop\_back();    std::cout << "Deque after popping: ";  for (const auto& elem : myDeque) {  std::cout << elem << " ";  }    std::cout << std::endl;  std::cout << "Size of deque: " << myDeque.size() << std::endl;    return 0;  } | Deque elements: 1 5 10 15  Front element: 1  Back element: 15  Deque after popping: 5 10  Size of deque: 2 |

# Priority Queue

# Heap

## What Is std::heap?

### Definition

std::heap is a **specialized tree-based structure** that satisfies the heap property. In a max-heap, for any given node, the value of the node is greater than or equal to the values of its children; in a min-heap, the value is less than or equal to the values of its children. Heaps are commonly used to implement priority queues.

### Memory Management

* **Dynamic Memory**: Heaps are typically implemented using dynamic arrays. The elements of the heap are stored in a contiguous memory location, making it efficient for access and storage.
* **Ownership**: The standard library does not provide a direct heap data structure like std::heap; instead, it provides functions that operate on ranges to manage heap properties.

### Similar Data Structures

* std::priority\_queue: This is a container that uses a heap to provide an efficient way to access the largest (or smallest) element.
* std::vector: Can be used to implement a heap through simple algorithms using the vector’s dynamic array capabilities.

## Pros and Cons

### Pros

* **Efficient Access**: Heaps allow for quick access to the maximum or minimum element, which is useful in priority queue implementations.
* **Dynamic Size**: Heaps can grow and shrink dynamically, utilizing memory efficiently.
* **Good Performance**: Insertion and deletion operations can be performed in O(log n) time, which is efficient for many applications.

### Cons

* **Complex Implementation**: Managing the heap property can be more complex than other data structures like arrays or linked lists.
* **Not Ordered**: While the maximum/minimum element can be accessed quickly, the rest of the elements are not stored in a sorted order.
* **Memory Overhead**: There may be additional memory overhead for managing pointers (if using a tree structure) or array indices.

## Iteration

The standard library does not provide a direct way to iterate through a heap because heaps are not guaranteed to maintain a specific order among non-root elements. However, you can use a vector to maintain the elements and then apply heap functions to manage the heap property.

### Using a Vector

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> heap = {10, 20, 30, 5, 15};

// Create a max-heap

std::make\_heap(heap.begin(), heap.end());

std::cout << "Heap elements: ";

for (const auto& elem : heap) {

std::cout << elem << " "; // Outputs the heap elements

}

std::cout << std::endl;

return 0;

}

## Built-in Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Parameters** | **Usage** | **Time** | **Space** | **Example** | **Reference Link** |
| make\_heap | RandomIt first, RandomIt last | Converts a range of elements into a heap. | O(n) | O(1) | std::make\_heap(myVector.begin(), myVector.end()); | [cplusplus](https://www.cplusplus.com/reference/algorithm/make_heap/) |
| push\_heap | RandomIt first, RandomIt last | Adds a new element to the heap. | O(logn) | O(1) | myVector.push\_back(25); std::push\_heap(myVector.begin(), myVector.end()); | [cplusplus](https://www.cplusplus.com/reference/algorithm/push_heap/) |
| pop\_heap | RandomIt first, RandomIt last | Removes the largest element from the heap. | O(logn) | O(1) | std::pop\_heap(myVector.begin(), myVector.end()); myVector.pop\_back(); | [cplusplus](https://www.cplusplus.com/reference/algorithm/pop_heap/) |
| sort\_heap | RandomIt first, RandomIt last | Sorts the elements in the heap. | O(nlog n) | O(1) | std::sort\_heap(myVector.begin(), myVector.end()); | [cplusplus](https://www.cplusplus.com/reference/algorithm/sort_heap/) |
| is\_heap | RandomIt first, RandomIt last | Checks if the range is a heap. | O(n) | O(1) | bool isHeap = std::is\_heap(myVector.begin(), myVector.end()); | [cplusplus](https://www.cplusplus.com/reference/algorithm/is_heap/) |
| is\_heap\_until | RandomIt first, RandomIt last | Returns the end of the heap range. | O(n) | O(1) | auto it = std::is\_heap\_until(myVector.begin(), myVector.end()); | [cplusplus](https://www.cplusplus.com/reference/algorithm/is_heap_until/) |

## Example

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> myHeap = {10, 20, 30, 5, 15};

// Create a max-heap

std::make\_heap(myHeap.begin(), myHeap.end());

std::cout << "Heap elements after make\_heap: ";

for (const auto& elem : myHeap) {

std::cout << elem << " "; // Outputs the heap elements

}

std::cout << std::endl;

// Add an element to the heap

myHeap.push\_back(25);

std::push\_heap(myHeap.begin(), myHeap.end());

std::cout << "Heap after push\_heap: ";

for (const auto& elem : myHeap) {

std::cout << elem << " ";

}

std::cout << std::endl;

// Remove the largest element

std::pop\_heap(myHeap.begin(), myHeap.end());

myHeap.pop\_back();

std::cout << "Heap after pop\_heap: ";

for (const auto& elem : myHeap) {

std::cout << elem << " ";

}

std::cout << std::endl;

// Sort the heap

std::sort\_heap(myHeap.begin(), myHeap.end());

std::cout << "Sorted elements: ";

for (const auto& elem : myHeap) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

# Ref

AI Prompt:

As a C++ master, can you teach me how to use the STL std::graph data structure in a concise and easy to understand way.

You must guide me very details about following points:

1. Section "What Is std::graph?": What is graph? How it manages its memory? Do we have other similar data structure like graph in STL C++?

2. Section "Pros and Cons": What is the pros and cons of graph? For each pros and cons, explain the reason. Describe in bullet points structure.

3. Section "Iteration": How to iterate through a graph? Give me all of the possible ways. And Give me example for each way. You example must be able to run without having me to edit anything.

4. Section "Built-in Functions": List for me all of the built-in functions that std::graph provides in a table format of 5 columns: function name and function parameter (include all the overloads), usage description, time complexity, space complexity, example, reference link (from cplusplus, make sure the link is not empty). Note that you can use any version of C++, including C++20 or below.

5. Section "Example": Finally, give me a big example that covers some of the most mostly-used built-in function of graph.