Die C++ STL

(Standard Template Library)

Inhaltsverzeichnis

Vektor (std::vector)

Beschreibung: Ein dynamisches Array, das Elemente in einem kontinuierlichen Speicherblock speichert.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen häufige Zugriffe auf Elemente über Indizes erforderlich sind und die meisten Einfügungen und Löschungen am Ende der Sequenz stattfinden.
- **Vorteile**:
  - Schneller Zugriff auf Elemente über Indizes (O(1)).
  - Effiziente Einfügungen und Löschungen am Ende (O(1) amortisiert).
- **Nachteile**:
  - Ineffiziente Einfügungen und Löschungen in der Mitte oder am Anfang (O(n)).

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    vec.push_back(6); // Einfügen am Ende
    vec[2] = 10; // Zugriff über Index
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

Methoden:

- **Konstruktoren und Destruktoren**:
  - `vector()`
  - `vector(size_type n)`
  - `vector(size_type n, const T& value)`
  - `vector(InputIterator first, InputIterator last)`
  - `vector(const vector& x)`
  - `vector(vector&& x)`
  - `vector(initializer_list<T> il)`
  - `~vector()`

- **Elementzugriff**:
  - `at(size_type n)`
  - `operator[](size_type n)`
  - `front()`
  - `back()`
  - `data()`

- **Iteratoren**:
  - `begin()`
  - `end()`
  - `rbegin()`
  - `rend()`
  - `cbegin()`
  - `cend()`
  - `crbegin()`
  - `crend()`

- **Kapazität**:
  - `empty()`
  - `size()`
  - `max_size()`
  - `reserve(size_type n)`
  - `capacity()`
  - `shrink_to_fit()`

- **Modifikatoren**:
  - `clear()`
  - `insert(const_iterator pos, const T& value)`
  - `insert(const_iterator pos, size_type n, const T& value)`
  - `insert(const_iterator pos, InputIterator first, InputIterator last)`
  - `insert(const_iterator pos, initializer_list<T> il)`
  - `erase(const_iterator pos)`
  - `erase(const_iterator first, const_iterator last)`
  - `push_back(const T& value)`
  - `push_back(T&& value)`
  - `pop_back()`
  - `resize(size_type n)`
  - `resize(size_type n, const T& value)`
  - `swap(vector& x)`

Deque `(std::deque)`

Beschreibung: Eine Double-Ended Queue, die Elemente in Blöcken speichert und effiziente Einfügungen und Löschungen an beiden Enden ermöglicht.

- **Verwendungszweck**: Ideal für Situationen, in denen häufige Einfügungen und Löschungen an beiden Enden erforderlich sind.
- **Vorteile**:
  - Effiziente Einfügungen und Löschungen an beiden Enden (O(1)).
  - Schneller Zugriff auf Elemente über Indizes (O(1)).
- **Nachteile**:
  - Weniger effizienter Zugriff auf mittlere Elemente im Vergleich zu `std::vector`.

```cpp
#include <iostream>
#include <deque>

int main() {
    std::deque<int> deq = {1, 2, 3, 4, 5};
    deq.push_back(6); // Einfügen am Ende
    deq.push_front(0); // Einfügen am Anfang
    for (int i : deq) {
        std::cout << i << " ";
    }
    return 0;
}
```

Methoden:

Konstruktoren und Destruktoren:

- `deque()`
- `deque(size_type n)`
- `deque(size_type n, const T& value)`
- `deque(InputIterator first, InputIterator last)`
- `deque(const deque& x)`
- `deque(deque&& x)`
- `deque(initializer_list<T> il)`
- `~deque()`

Elementzugriff:

- `at(size_type n)`
- `operator[](size_type n)`
- `front()`
- `back()`

- **Iteratoren**:
  - `begin()`
  - `end()`
  - `rbegin()`
  - `rend()`
  - `cbegin()`
  - `cend()`
  - `crbegin()`
  - `crend()`

- **Kapazität**:
  - `empty()`
  - `size()`
  - `max_size()`
  - `shrink_to_fit()`

- **Modifikatoren**:
  - `clear()`
  - `insert(const_iterator pos, const T& value)`
  - `insert(const_iterator pos, size_type n, const T& value)`
  - `insert(const_iterator pos, InputIterator first, InputIterator last)`
  - `insert(const_iterator pos, initializer_list<T> il)`
  - `erase(const_iterator pos)`
  - `erase(const_iterator first, const_iterator last)`
  - `push_back(const T& value)`
  - `push_back(T&& value)`
  - `pop_back()`
  - `push_front(const T& value)`
  - `push_front(T&& value)`
  - `pop_front()`
  - `resize(size_type n)`
  - `resize(size_type n, const T& value)`
  - `swap(deque& x)`

List (std::list)

Beschreibung: Eine doppelt verkettete Liste, die Elemente in Knoten speichert, die aufeinander verweisen.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen häufige Einfügungen und Löschungen in der Mitte der Sequenz erforderlich sind.
- **Vorteile**:
  - Effiziente Einfügungen und Löschungen an beliebigen Positionen (O(1)).
- **Nachteile**:
  - Langsamer Zugriff auf Elemente über Indizes (O(n)).

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> lst = {1, 2, 3, 4, 5};
    lst.push_back(6); // Einfügen am Ende
    lst.push_front(0); // Einfügen am Anfang
    auto it = lst.begin();
    ++it; // Auf das zweite Element zeigen
    lst.insert(it, 10); // Einfügen in der Mitte
    for (int i : lst) {
        std::cout << i << " ";
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
  - `list()`
  - `list(size_type n)`
  - `list(size_type n, const T& value)`
  - `list(InputIterator first, InputIterator last)`
  - `list(const list& x)`
  - `list(list&& x)`
  - `list(initializer_list<T> il)`
  - `~list()`

- **Elementzugriff**:

    - front()
    - back()

- **Iteratoren**:
    - begin()
    - end()
    - rbegin()
    - rend()
    - cbegin()
    - cend()
    - crbegin()
    - crend()

- **Kapazität**:
    - empty()
    - size()
    - max_size()

- **Modifikatoren**:
    - clear()
    - insert(const_iterator pos, const T& value)
    - insert(const_iterator pos, size_type n, const T& value)
    - insert(const_iterator pos, InputIterator first, InputIterator last)
    - insert(const_iterator pos, initializer_list<T> il)
    - erase(const_iterator pos)
    - erase(const_iterator first, const_iterator last)
    - push_back(const T& value)
    - push_back(T&& value)
    - pop_back()
    - push_front(const T& value)
    - push_front(T&& value)
    - pop_front()
    - resize(size_type n)
    - resize(size_type n, const T& value)
    - swap(list& x)
    - merge(list& x)
    - merge(list&& x)
    - splice(const_iterator pos, list& x)
    - splice(const_iterator pos, list&& x)
    - splice(const_iterator pos, list& x, const_iterator i)
    - splice(const_iterator pos, list&& x, const_iterator i)
    - splice(const_iterator pos, list& x, const_iterator first, const_iterator last)

- splice(const_iterator pos, list&& x, const_iterator first, const_iterator last)
- remove(const T& value)
- remove_if(Predicate pred)
- unique()
- unique(BinaryPredicate binary_pred)
- sort()
- sort(Compare comp)
- reverse()

Forward List (std::forward_list)

Beschreibung: Eine einfach verkettete Liste, die Elemente in Knoten speichert, die auf den nächsten Knoten verweisen.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen Speicherplatz gespart werden soll und Einfügungen und Löschungen an beliebigen Positionen erforderlich sind.
- **Vorteile**:
  - Effiziente Einfügungen und Löschungen an beliebigen Positionen (O(1)).
  - Weniger Speicherverbrauch im Vergleich zu `std::list`.
- **Nachteile**:
  - Langsamer Zugriff auf Elemente über Indizes (O(n)).
  - Kein direkter Zugriff auf vorherige Elemente.

```
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> flst = {1, 2, 3, 4, 5};
    flst.push_front(0); // Einfügen am Anfang
    auto it = flst.begin();
    ++it; // Auf das zweite Element zeigen
    flst.insert_after(it, 10); // Einfügen nach dem zweiten
Element
    for (int i : flst) {
        std::cout << i << " ";
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren**:

- `forward_list()`
- `forward_list(size_type n)`
- `forward_list(size_type n, const T& value)`
- `forward_list(InputIterator first, InputIterator last)`
- `forward_list(const forward_list& x)`
- `forward_list(forward_list&& x)`
- `forward_list(initializer_list<T> il)`
- `~forward_list()`

- **Elementzugriff**:
  - `front()`

- **Iteratoren**:
  - `begin()`
  - `end()`
  - `cbegin()`
  - `cend()`
  - `before_begin()`
  - `cbefore_begin()`

- **Kapazität**:
  - `empty()`
  - `max_size()`

- **Modifikatoren**:
  - `clear()`
  - `insert_after(const_iterator pos, const T& value)`
  - `insert_after(const_iterator pos, size_type n, const T& value)`
  - `insert_after(const_iterator pos, InputIterator first, InputIterator last)`
  - `insert_after(const_iterator pos, initializer_list<T> il)`
  - `erase_after(const_iterator pos)`
  - `erase_after(const_iterator first, const_iterator last)`
  - `push_front(const T& value)`
  - `push_front(T&& value)`
  - `pop_front()`
  - `resize(size_type n)`
  - `resize(size_type n, const T& value)`
  - `swap(forward_list& x)`
  - `splice_after(const_iterator pos, forward_list& x)`
  - `splice_after(const_iterator pos, forward_list&& x)`
  - `splice_after(const_iterator pos, forward_list& x, const_iterator i)`
  - `splice_after(const_iterator pos, forward_list&& x, const_iterator i)`
  - `splice_after(const_iterator pos, forward_list& x, const_iterator first, const_iterator last)`
  - `splice_after(const_iterator pos, forward_list&& x, const_iterator first, const_iterator last)`
  - `remove(const T& value)`
  - `remove_if(Predicate pred)`
  - `unique()`

- unique(BinaryPredicate binary_pred)
- sort()
- sort(Compare comp)
- reverse()

Set (std::set)

Beschreibung: Eine Menge von Elementen, die automatisch sortiert und eindeutig sind.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine sortierte Sammlung eindeutiger Elemente benötigt wird.
- **Vorteile**:
  - Automatische Sortierung und Eindeutigkeit der Elemente.
  - Effiziente Einfügungen, Löschungen und Suchen (O(log n)).
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> s = {5, 3, 1, 4, 2};
    s.insert(6); // Einfügen
    s.erase(3); // Löschen
    for (int i : s) {
        std::cout << i << " ";
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
  - `set()`
  - `set(InputIterator first, InputIterator last)`
  - `set(const set& x)`
  - `set(set&& x)`
  - `set(initializer_list<T> il)`
  - `~set()`

- **Iteratoren**:
  - `begin()`
  - `end()`
  - `rbegin()`
  - `rend()`
  - `cbegin()`
  - `cend()`
  - `crbegin()`
  - `crend()`

- **Kapazität**:
  - `empty()`
  - `size()`
  - `max_size()`

- **Modifikatoren**:
  - `clear()`
  - `insert(const value_type& value)`
  - `insert(value_type&& value)`
  - `insert(InputIterator first, InputIterator last)`
  - `insert(initializer_list<value_type> il)`
  - `emplace(Args&&... args)`
  - `emplace_hint(const_iterator hint, Args&&... args)`
  - `erase(const_iterator pos)`
  - `erase(const key_type& key)`
  - `erase(const_iterator first, const_iterator last)`
  - `swap(set& x)`
  - `merge(set& source)`
  - `merge(set&& source)`
  - `extract(const_iterator pos)`
  - `extract(const key_type& key)`
  - `insert(node_type&& nh)`

- **Suchen**:
  - `find(const key_type& key)`
  - `count(const key_type& key)`
  - `lower_bound(const key_type& key)`
  - `upper_bound(const key_type& key)`
  - `equal_range(const key_type& key)`

Multiset (std::multiset)

Beschreibung: Eine Menge von Elementen, die automatisch sortiert sind, aber Duplikate erlaubt.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine sortierte Sammlung von Elementen benötigt wird, die Duplikate enthalten kann.
- **Vorteile**:
  - Automatische Sortierung der Elemente.
  - Effiziente Einfügungen, Löschungen und Suchen (O(log n)).
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <set>

int main() {
    std::multiset<int> ms = {5, 3, 1, 4, 2, 3};
    ms.insert(6); // Einfügen
    ms.erase(3); // Löschen
    for (int i : ms) {
        std::cout << i << " ";
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren:**
  - `multiset()`
  - `multiset(InputIterator first, InputIterator last)`
  - `multiset(const multiset& x)`
  - `multiset(multiset&& x)`
  - `multiset(initializer_list<T> il)`
  - `~multiset()`

- **Iteratoren**:
  - `begin()`
  - `end()`
  - `rbegin()`
  - `rend()`
  - `cbegin()`
  - `cend()`
  - `crbegin()`
  - `crend()`

- **Kapazität**:
  - `empty()`
  - `size()`
  - `max_size()`

- **Modifikatoren**:
  - `clear()`
  - `insert(const value_type& value)`
  - `insert(value_type&& value)`
  - `insert(InputIterator first, InputIterator last)`
  - `insert(initializer_list<value_type> il)`
  - `emplace(Args&&... args)`
  - `emplace_hint(const_iterator hint, Args&&... args)`
  - `erase(const_iterator pos)`
  - `erase(const key_type& key)`
  - `erase(const_iterator first, const_iterator last)`
  - `swap(multiset& x)`
  - `merge(multiset& source)`
  - `merge(multiset&& source)`
  - `extract(const_iterator pos)`
  - `extract(const key_type& key)`
  - `insert(node_type&& nh)`

- **Suchen**:
  - `find(const key_type& key)`
  - `count(const key_type& key)`
  - `lower_bound(const key_type& key)`
  - `upper_bound(const key_type& key)`
  - `equal_range(const key_type& key)`

Map (std::map)

Beschreibung: Eine assoziative Sammlung von Schlüssel-Wert-Paaren, die automatisch nach den Schlüsseln sortiert ist.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine sortierte Sammlung von Schlüssel-Wert-Paaren benötigt wird.
- **Vorteile**:
  - Automatische Sortierung nach den Schlüsseln.
  - Effiziente Einfügungen, Löschungen und Suchen (O(log n)).
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<int, std::string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
    m[4] = "four"; // Einfügen
    m.erase(2); // Löschen
   for (const auto& pair : m) {
       std::cout << pair.first << ": " << pair.second << std::endl;
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
- `map()`
- `map(InputIterator first, InputIterator last)`
- `map(const map& x)`
- `map(map&& x)`
- `map(initializer_list<pair<const Key, T>> il)`
- `~map()`
- **Elementzugriff**:
- `at(const key_type& key)`
- `operator[](const key_type& key)`

- **Iteratoren**:

- `begin()`
- `end()`
- `rbegin()`
- `rend()`
- `cbegin()`
- `cend()`
- `crbegin()`
- `crend()`

- **Kapazität**:

- `empty()`
- `size()`
- `max_size()`

- **Modifikatoren**:

- `clear()`
- `insert(const value_type& value)`
- `insert(value_type&& value)`
- `insert(InputIterator first, InputIterator last)`
- `insert(initializer_list<value_type> il)`
- `emplace(Args&&... args)`
- `emplace_hint(const_iterator hint, Args&&... args)`
- `erase(const_iterator pos)`
- `erase(const key_type& key)`
- `erase(const_iterator first, const_iterator last)`
- `swap(map& x)`
- `merge(map& source)`
- `merge(map&& source)`
- `extract(const_iterator pos)`
- `extract(const key_type& key)`
- `insert(node_type&& nh)`

- **Suchen**:

- `find(const key_type& key)`
- `count(const key_type& key)`
- `lower_bound(const key_type& key)`
- `upper_bound(const key_type& key)`
- `equal_range(const key_type& key)`

Multimap (std::multimap)

Beschreibung: Eine assoziative Sammlung von Schlüssel-Wert-Paaren, die automatisch nach den Schlüsseln sortiert ist und Duplikate erlaubt.

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine sortierte Sammlung von Schlüssel-Wert-Paaren benötigt wird, die Duplikate enthalten kann.
- **Vorteile**:
  - Automatische Sortierung nach den Schlüsseln.
  - Effiziente Einfügungen, Löschungen und Suchen (O(log n)).
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```
#include <iostream>
#include <map>

int main() {
    std::multimap<int, std::string> mm = {{1, "one"}, {2,
"two"}, {2, "two"}, {3, "three"}};
    mm.insert({4, "four"}); // Einfügen
    mm.erase(2); // Löschen
    for (const auto& pair : mm) {
        std::cout << pair.first << ": " << pair.second <<
std::endl;
    }
    return 0;
}
```

- **Konstruktoren und Destruktoren**:

- `multimap()`
- `multimap(InputIterator first, InputIterator last)`
- `multimap(const multimap& x)`
- `multimap(multimap&& x)`
- `multimap(initializer_list<pair<const Key, T>> il)`
- `~multimap()`

- **Iteratoren**:

- `begin()`
- `end()`
- `rbegin()`
- `rend()`
- `cbegin()`
- `cend()`
- `crbegin()`
- `crend()`

- **Kapazität**:

- `empty()`
- `size()`
- `max_size()`

- **Modifikatoren**:

- `clear()`
- `insert(const value_type& value)`
- `insert(value_type&& value)`
- `insert(InputIterator first, InputIterator last)`
- `insert(initializer_list<value_type> il)`
- `emplace(Args&&... args)`
- `emplace_hint(const_iterator hint, Args&&... args)`
- `erase(const_iterator pos)`
- `erase(const key_type& key)`
- `erase(const_iterator first, const_iterator last)`
- `swap(multimap& x)`
- `merge(multimap& source)`
- `merge(multimap&& source)`
- `extract(const_iterator pos)`
- `extract(const key_type& key)`
- `insert(node_type&& nh)`

- **Suchen**:

- `find(const key_type& key)`
- `count(const key_type& key)`
- `lower_bound(const key_type& key)`
- `upper_bound(const key_type& key)`
- `equal_range(const key_type& key)`

Stack (std::stack)

Beschreibung: Ein LIFO (Last In, First Out) Container-Adapter, der auf einem anderen Container basiert (standardmäßig std::deque).

- **Verwendungszweck**: Gut geeignet für Situationen, in denen ein Stapel benötigt wird, bei dem das letzte eingefügte Element zuerst entfernt wird.
- **Vorteile**:
  - Einfache Implementierung eines Stapels.
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> stk;
    stk.push(1); // Einfügen
    stk.push(2); // Einfügen
    stk.push(3); // Einfügen
    std::cout << stk.top() << std::endl; // Zugriff auf das
oberste Element
    stk.pop(); // Entfernen des obersten Elements
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
- `stack()`
- `stack(const Container& cont)`
- `stack(Container&& cont)`
- `~stack()`
- **Elementzugriff**:
- `top()`

- **Kapazität**:
- `empty()`
- `size()`

- **Modifikatoren**:
- `push(const value_type& value)`
- `push(value_type&& value)`
- `pop()`
- `swap(stack& x)`
- `emplace(Args&&... args)`

Queue `(std::queue)`

Beschreibung: Ein FIFO (First In, First Out) Container-Adapter, der auf einem anderen Container basiert (standardmäßig std::deque).

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine Warteschlange benötigt wird, bei der das erste eingefügte Element zuerst entfernt wird.
- **Vorteile**:
  - Einfache Implementierung einer Warteschlange.
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1); // Einfügen
    q.push(2); // Einfügen
    q.push(3); // Einfügen
    std::cout << q.front() << std::endl; // Zugriff auf das vorderste Element
    q.pop(); // Entfernen des vordersten Elements
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
  - `queue()`
  - `queue(const Container& cont)`
  - `queue(Container&& cont)`
  - `~queue()`
- **Elementzugriff**:
  - `front()`
  - `back()`
- **Kapazität**:
  - `empty()`
  - `size()`

- **Modifikatoren**:
    - `push(const value_type& value)`
    - `push(value_type&& value)`
    - `pop()`
    - `swap(queue& x)`
    - `emplace(Args&&... args)`

Priority Queue (std::priority_queue)

Beschreibung: Ein Container-Adapter, der eine Prioritätswarteschlange implementiert, bei der das größte Element zuerst entfernt wird (standardmäßig basierend auf std::vector).

- **Verwendungszweck**: Gut geeignet für Situationen, in denen eine Prioritätswarteschlange benötigt wird, bei der das größte Element zuerst entfernt wird.
- **Vorteile**:
  - Einfache Implementierung einer Prioritätswarteschlange.
- **Nachteile**:
  - Kein direkter Zugriff auf Elemente über Indizes.

```cpp
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(1); // Einfügen
    pq.push(3); // Einfügen
    pq.push(2); // Einfügen
    std::cout << pq.top() << std::endl; // Zugriff auf das größte Element
    pq.pop(); // Entfernen des größten Elements
    return 0;
}
```

- **Konstruktoren und Destruktoren**:
  - `priority_queue()`
  - `priority_queue(const Compare& comp)`
  - `priority_queue(const Compare& comp, const Container& cont)`
  - `priority_queue(const Compare& comp, Container&& cont)`
  - `priority_queue(const priority_queue& x)`
  - `priority_queue(priority_queue&& x)`
  - `~priority_queue()`

- **Elementzugriff**:
  - `top()`

- **Kapazität**:
    - `empty()`
    - `size()`

- **Modifikatoren**:
    - `push(const value_type& value)`
    - `push(value_type&& value)`
    - `pop()`
    - `swap(priority_queue& x)`
    - `emplace(Args&&... args)`