Container

C++ Container (세부 설명)

C++에서 **Container**(컨테이너)는 데이터를 저장하고 관리하는 데 사용되는 자료구조를 말합니다. C++ 표준 라이브러리(STL: Standard Template Library)는 다양한 컨테이너를 제공하며, 각 컨테이너는 데이터의 저장 방식, 접근 속도, 삭제/삽입/탐색 성능, 메모리 사용량 등에 따라 특성과 용도가 다릅니다.

이 문서는 C++에서 자주 사용되는 **Set, Vector, List, LinkedList, deque, stack, queue, map, unordered_map** 등 주요 컨테이너들을 **구조, 특성, 시간 복잡도, 사용 예시, 실무 팁**까지 세세하게 설명합니다. 각 컨테이너는 문제 해결에 적합한 자료구조를 제공하며, 개발자가 문제의 특성에 따라 적절한 컨테이너를 선택할 수 있도록 하기 위해 자세한 분석을 제공합니다.

---

1. Vector (동적 배열)

▶ 구조

- `std::vector<T>`는 동적 배열로, `T` 타입의 요소들을 연속된 메모리 공간에 저장합니다.
- 크기를 고정하지 않고, 필요할 때 자동으로 확장합니다.
- 내부적으로 `capacity`와 `size`를 관리합니다.

▶ 특징

항목	설명	
접근 속도	O(1) (인덱스 기반)	
삽입/삭제	O(n) (끝에 삽입/삭제는 O(1), 중간 삽입/삭제는 O(n))	
메모리 효율	연속된 메모리 → 캐시 효율성 우수	
자동 확장	`push_back`, `pop_back` 등으로 확장 가능	

▶ 시간 복잡도 (주요 연산)

연산	시간 복잡도
생성	O(1)
push_back	O(1) (평균), O(n) (최악)
pop_back	O(1)
at(i)	O(1)
insert(i, x)	O(n)

연산	시간 복잡도
erase(i)	O(n)
find(x)	O(n)
sort()	O(n log n)

▶ 예시 코드 (C++)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> vec = {1, 3, 5, 7, 9};
    vec.push_back(11);  // 뒤에 추가
    vec.insert(vec.begin() + 2, 4);  // 중간 삽입
    vec.erase(vec.begin() + 1);      // 중간 삭제

    for (int x : vec) {
        cout << x << " ";
    }
    cout << endl;

    // 정렬
    sort(vec.begin(), vec.end());
    return 0;
}
```

▶ 실무 팁

- **빈도 높은 접근** (예: 배열 순회) → `vector` 가 최적
- **중간 삽입/삭제가 자주 발생** → `list` 나 `deque` 를 고려
- **메모리 캐시 효율** → 연속 메모리 → 빠른 접근

---

2. List (연결 리스트)

▶ 구조

- `std::list<T>` 는 **이중 연결 리스트**(doubly linked list) 구조를 사용합니다.
- 각 노드는 `prev` , `next` 포인터를 포함하며, 순서를 유지합니다.
- 메모리가 연속적이지 않음.

▶ 특징

항목	설명
접근 속도	O(n) (인덱스 접근 불가)
삽입/삭제	O(1) (중간/끝/시작 모두 가능)
메모리 효율	낮음 (포인터 추가)
순회	O(n)

▶ 시간 복잡도 (주요 연산)

연산	시간 복잡도
push_front	O(1)
push_back	O(1)
pop_front	O(1)
pop_back	O(1)
insert(pos, x)	O(1) (pos가 지정되면)
erase(pos)	O(1)
find(x)	O(n)
size()	O(n)

▶ 예시 코드

```
#include <iostream>
#include <list>
using namespace std;

int main() {
    list<int> lst = {1, 2, 3, 4, 5};

    // 앞에 삽입
    lst.push_front(0);
    // 뒤에 삽입
    lst.push_back(6);

    // 중간 삽입
    auto it = lst.begin();
    advance(it, 2);
    lst.insert(it, 100);

    // 순회
    for (int x : lst) {
        cout << x << " ";
    }
```

```
    cout << endl;

    return 0;
}
```

▶ 실무 팁

- **중간/끝 삽입/삭제가 자주 발생** → `list`가 최적
- **인덱스 기반 접근 필요 없음** → `vector`보다 적합
- **순회가 자주 필요** → `list`는 순회에 유리

---

3. Deque (Double-ended Queue)

▶ 구조

- `std::deque<T>`는 **양 끝에서 삽입/삭제가 가능한 큐**입니다.
- `vector`와 `list`의 장점을 결합한 자료구조.
- 메모리가 연속적이지 않지만, 끝에만 효율적으로 접근 가능.

▶ 특징

항목	설명
접근 속도	O(1) (앞/뒤)
삽입/삭제	O(1) (앞/뒤), O(n) (중간)
메모리	중간에 삽입 시 비효율적
크기 제한	없음 (무한 확장 가능)

▶ 시간 복잡도

연산	시간 복잡도
push_front	O(1)
push_back	O(1)
pop_front	O(1)
pop_back	O(1)
insert(i, x)	O(n) (중간)
find(x)	O(n)
size()	O(1)

▶ 예시 코드

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {
    deque<int> dq;
    dq.push_back(1);
    dq.push_front(0);
    dq.push_back(2);

    for (int x : dq) {
        cout << x << " ";
    }
    cout << endl;

    // 중간 삽입 (비효율적)
    dq.insert(dq.begin() + 1, 50);
    return 0;
}
```

▶ 실무 팁

- **양 끝에서 자주 삽입/삭제** (예: 슬라이딩 윈도우, BFS) → `deque`가 최적
- **중간 삽입은 피해야 함** → `vector`나 `list`보다 느림

---

4. Set (집합)

▶ 구조

- `std::set<T>`는 **자동 정렬된 순서를 가지는 집합**입니다.
- `T` 타입이 **비교 가능**해야 하며, `std::less<T>` 기반으로 정렬.
- 중복 허용 없음 (unique).

▶ 특징

항목	설명
접근 속도	O(log n) (find, insert, erase)
중복 제거	자동
순서	정렬된 순서 (예: 1, 2, 3, 4)
메모리	O(n)

▶ 시간 복잡도

연산	시간 복잡도
insert(x)	O(log n)
find(x)	O(log n)
erase(x)	O(log n)
size()	O(1)
empty()	O(1)

▶ 예시 코드

```cpp
#include <iostream>
#include <set>
using namespace std;

int main() {
    set<int> s = {5, 2, 8, 1, 9};
    s.insert(3);  // 중복 방지
    s.erase(8);

    for (int x : s) {
        cout << x << " ";
    }
    cout << endl;

    return 0;
}
```

▶ 실무 팁

- **중복 제거 + 정렬 필요** → `set` 이 최적
- **순서가 중요** (예: 정렬된 데이터 처리) → `set` 사용
- `unordered_set` 으로 **해시 기반**으로 더 빠르게 처리 가능

---

5. Multiset (중복 허용 집합)

▶ 구조

- `std::multiset<T>` 는 `set` 과 유사하지만 **중복 허용**합니다.
- 정렬 유지 + 중복 허용

▶ 특징

항목	설명
중복 허용	O(1)
순서	정렬된 순서 유지
시간 복잡도	O(log n)

▶ 예시

```cpp
#include <iostream>
#include <multiset>
using namespace std;

int main() {
    multiset<int> ms = {1, 2, 2, 3, 3, 3};
    ms.insert(4);
    ms.erase(ms.find(2));  // 특정 값 제거

    for (int x : ms) {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}
```

---

6. Map (키-값 맵)

▶ 구조

- `std::map<K, V>` 는 **키 기반으로 값을 저장**하는 자료구조.
- `K` 는 **비교 가능**해야 하며, `std::less<K>` 기반 정렬.
- 자동 정렬, 중복 허용 없음.

▶ 특징

항목	설명
접근 속도	O(log n) (find, insert, erase)
중복 키	허용 안 됨
순서	정렬된 순서 유지

▶ 시간 복잡도

연산	시간 복잡도
insert(key, value)	O(log n)
find(key)	O(log n)
erase(key)	O(log n)
size()	O(1)

▶ 예시

```cpp
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<string, int> mp;
    mp["apple"] = 5;
    mp["banana"] = 3;
    mp["cherry"] = 8;

    for (auto& [k, v] : mp) {
        cout << k << ": " << v << endl;
    }
    return 0;
}
```

---

7. Unordered Map (해시 맵)

▶ 구조

- `std::unordered_map<K, V>` 는 **해시 테이블 기반**으로 저장.
- 평균 O(1) 시간 복잡도 (최악 O(n))

▶ 특징

항목	설명
접근 속도	평균 O(1), 최악 O(n)
순서	무순 (임의 순서)
중복 키	허용 안 됨 (단, `unordered_multimap` 있음)

▶ 시간 복잡도

연산	시간 복잡도
insert	O(1) (평균)
find	O(1) (평균)
erase	O(1) (평균)
size()	O(1)

▶ 예시

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

int main() {
    unordered_map<string, int> um;
    um["a"] = 1;
    um["b"] = 2;
    um["c"] = 3;

    for (auto& [k, v] : um) {
        cout << k << ": " << v << endl;
    }
    return 0;
}
```

---

8. Stack (스택)

▶ 구조

- `std::stack<T>` 는 **LIFO**(Last In, First Out) 구조.
- `push` , `pop` , `top` , `size` , `empty` 연산 제공.

▶ 특징

항목	설명
접근 방식	LIFO
시간 복잡도	O(1) (모든 연산)
자료구조	일반적으로 `vector` 또는 `deque` 기반

▶ 예시

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> st;
    st.push(1);
    st.push(2);
    st.push(3);

    while (!st.empty()) {
        cout << st.top() << " ";
        st.pop();
    }
    return 0;
}
```

---

9. Queue (큐)

▶ 구조

- `std::queue<T>` 는 **FIFO**(First In, First Out) 구조.
- `push` , `pop` , `front` , `back` , `size` , `empty` 제공.

▶ 특징

항목	설명
접근 방식	FIFO
시간 복잡도	O(1) (모든 연산)
자료구조	`deque` 기반 (효율적)

▶ 예시

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;
    q.push(1);
    q.push(2);
```

```
    q.push(3);

    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

---

10. Priority Queue (우선순위 큐)

▶ 구조

- `std::priority_queue<T>` 는 **최대/최소 우선순위**를 기반으로 저장.
- 최대 힙 (default), 최소 힙 가능 (reverse)

▶ 특징

항목	설명
접근 방식	우선순위 기반
시간 복잡도	O(log n) (삽입/삭제)
자료구조	최대 힙 (default)

▶ 예시

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(10);
    pq.push(5);
    pq.push(15);

    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    return 0;
}
```

컨테이너 비교 요약 (표)

컨테이너	접근	삽입/삭제	중복	정렬	메모리
vector	O(1)	O(n) (중간)	❌	✅	중간
list	O(n)	O(1)	❌	✅	높음 (포인터)
deque	O(1) (앞/뒤)	O(1) (앞/뒤), O(n) (중간)	❌	✅	중간
set	O(log n)	O(log n)	❌	✅	중간
multiset	O(log n)	O(log n)	✅	✅	중간
map	O(log n)	O(log n)	❌	✅	중간
unordered_map	O(1)	O(1)	❌	❌	중간
stack	O(1)	O(1)	❌	❌	낮음
queue	O(1)	O(1)	❌	❌	낮음

실무에서의 선택 기준

상황	추천 컨테이너
순회 + 빠른 접근	`vector`
중간 삽입/삭제 자주 발생	`list`
양 끝 삽입/삭제	`deque`
중복 제거 + 정렬	`set`, `map`
빠른 키-값 접근	`unordered_map`
스택/큐 구현	`stack`, `queue`
최대/최소 찾기	`priority_queue`

실무 팁 & 주의사항

1. **메모리 관리**: `vector`는 `new`보다 더 안전하고 자동 해제 가능.
2. **순서 중요 여부**: 정렬이 필요하면 `set`, `map` 사용. 순서 없으면 `unordered_map`.
3. **성능 최적화**: `unordered_map`은 평균 O(1)이지만, 충돌 시 O(n)이므로 `hash` 크기 조절 필요.
4. **예외 처리**: `vector`의 `at(i)`는 범위 초과 시 예외 발생 (안전), `operator[]`는 범위 초과 시 undefined behavior.
5. **C++11 이상 권장**: `auto`, `range-based for`, `std::pair` 등 최신 기능 활용.

결론

C++의 STL 컨테이너는 **문제의 특성에 따라 최적의 자료구조를 선택**하는 데 핵심적인 역할을 합니다.

- **접근 빈도** → `vector`
- **삽입/삭제 빈도** → `list`, `deque`
- **정렬/중복 제거** → `set`, `map`
- **빠른 키-값 접근** → `unordered_map`
- **스택/큐** → `stack`, `queue`

개발자는 **문제의 요구사항** (예: 중복 허용 여부, 순서, 성능)을 파악하고, 적절한 컨테이너를 선택함으로써 **효율성과 안정성**을 극대화할 수 있습니다.

참고 자료

- C++ Reference: https://en.cppreference.com
- STL 문서: https://www.cplusplus.com/reference/
- "C++ Primer" (5th Edition) – Chapter 18 (STL)
- "Effective C++" – Item 31: Use the Right Container
