

## • <chrono> : 시간 관련 라이브러리

- clock 클래스

✓ system\_clock : 기본 clock이며, 시스템의 시간을 나타냄. C의 time\_t와 호환 가능.

✓ steady\_clock : 시점 영향이 불가능한 clock. 정확한 현재 시간을 이용할 필요가 있는 경우 이용.

✓ high\_resolution\_clock : Windows / Linux에서 제공하는 clock. 정밀도 가장 높음.

✓ 주요 메소드

→ now() : clock의 현재 시간을 나타내는 time\_point를 반환

→ operator +, - : duration을 반환.

→ duration\_cast<duration 지원 시간 단위>(duration) : 템플릿의 시간 단위로 duration을 변환하여 반환. (<chrono>의 메소드)

→ count() : duration 지원 시간 단위 값을 int로 반환. (std::duration의 메소드)

- std::clock::time\_point : 시간의 한 층(세대)을 표현.

- std::chrono::duration : 시간의 한 기간을 표현

- duration은 hours ~ nanoseconds 단위 까지 지원.

## • Standard Libraries - Containers

- sequence Container : 객체를 순차적으로 보관.

✓ <vector> : 가변형이 배열

→ 주요 메소드

- operator [] (int), at(int) : 인자 위치의 원소 반환

- push\_back(T), pop\_back(T) : 가장 뒤에 원소 삽입 / 삭제 ( $O(1)$ 에 해결 가능)

- size() : vector의 크기 (내부의 원소 개수)를 vector::size\_type으로 반환.

- insert (vector::iterator, T) : iterator 위치에 원소 삽입 (최대  $O(n)$ )

- erase (vector::iterator) : iterator 위치 값은 삭제 (최대  $O(n)$ )

- begin() : 첫번째 원소를 가리키는 iterator 반환.

- `end()`: 마지막 원소 '다음 원소' iterator 반환.
- `cbegin(), rbegin(), crbegin()` ...: C의 경우 iterator를 통해 값을 변경할 수 있는 iterator 반환, `r`은 반대 끝에서 시작하여 `++`하면 역으로 이동하는 iterator 반환. `end()`의 경우도 동일하게 처리!

→ `vector`의 index는 부호없는 정수 타입이기 때문에 `vector<T>::size_type i`의 값이 0일 때 `--`을 수행하면 -1이 아니라 가장 큰 정수 타입 값이 반환된다.

### ✓ <list>: Doubly Linked-List의 구현체

→ `vector`와 달리 `operator []()`나 `at()`을 통해 원소에 접근할 수 있으나, 삽입, 삭제를  $O(1)$ 에 수행할 수 있다.

→ `list::iterator`는 Bidirectional Iterator 타입이라 `vector`의 RandomAccessIterator와 달리 한번에 한 칸씩 뒤에 옮길 수 있는 특성이기 때문에 (`operator ++()`, `operator --()`만 이용 가능) 힙색의 효율은 좋지 않다. (메모리 상 연속되지 않은 공간에 원소들이 배치되어 있기 때문!)

→ `insert`, `erase` 이후에도 iterator가 유효하며, `insert`는 삽입된 첫 값을 가리키는 iterator, `erase`는 삭제된 다음 값의 iterator (즉, 삭제된 index의 위치)를 반환한다.

### ✓ <deque>: Double-Ended Queue의 구현체

→ 양의 위치 원소 접근 및 삽입/삭제는 `vector`와 동일하게 각각  $O(1)$  /  $O(n)$ .

→ 맨 앞 / 맨 뒤에 원소 삽입 / 삭제를 지원하여 그 속도도  $O(1)$ 로 지원!

• `deque`는 원소들을 실제 저장하는 Block들의 주소값 배열로 구현되어 있으며, 처음 생성시 Block 주소값 배열의 앞/뒤를 어느정도 비워둘 때 생성하기에  $O(1)$  가능. 큰 주소값 배열이 가득차면 새 배열로 주소값들을 옮기는 대역간 시장이 소요되며, 그래도 주소값만 옮기면 되어서 `vector`보다 빠르기 동작.

→ `deque`는 일반적으로 `vector`보다 속도가 빠르나, Block 주소값 저장 등을 위하여 메모리를 더 많이 차지한다.

→ `deque::iterator`는 `vector`와 동일한 RandomAccessIterator 타입이라 `operator []()`, `at()`을 통해 양의 원소 접근이 가능하며, 그 동작 방식도 `vector`와 동일함.

→ 주요 메소드

- `push_front(T)`: 맨 앞에 원소 삽입
- `push_back(T)`: 맨 뒤에 원소 삽입
- `pop_front(T)`: 맨 앞 원소 삭제
- `pop_back(T)`: 맨 뒤 원소 삭제

변환값 void.

### ✓ 어떤 sequence container를 사용해야 하는가?

→ 일반적인 상황에서는 거의 만들이라 할 수 있는 `vector`를 사용한다.

→ 원소들을 순차적으로만 접근하여, 리스트의 중간에 원소 삽입/삭제가 자주 일어난다면 `list`을 사용한다.

→ stack, queue 음울과 같이 리스트의 양 끝에서 원소 삽입/삭제가 자주 일어나다면 deque를 사용한다.

- Associative Container : Key - Value 구조를 가지는 Container.

✓ Key의 존재 유무만 필요할 때는 set, value까지 필요할 때는 map을 쓰며, map 안으로도 set의 역할을 수행할 수 있으나 map이 메모리를 더 많이 차지하기 때문에 key만 필요하면 set을 쓰는 것이 좋다.

✓ <set> : 집합, key만 저장하고, key 간에는 특히 순서 없음. 템플릿 인자는 Key의 타입.

→ insert 시 (T key) 정보만 주고 어디에 삽입할 지의 정보는 주지 않음.

→ Associative Container은 Red-Black Tree로 구현되어 있어 탐색/삽입/삭제가  $O(\log n)$ 으로 가능.

→ set의 iterator는 Bidirectional Iterator 타입이라 임의의 위치 접근은 불가하고 순차적으로 하나씩만 접근할 수 있다. 또한 범위기반 for문도 사용할 수 있다!

→ 내부 원소들은 정렬된 상태를 유지한다. (Red-Black Tree 이용)

→ 집합의 구현체이기 때문에 중복된 원소가 존재하지 않는다.

→ 사용자 정의 타입을 key로 이용하기 위해서는 크게 2가지 (operator <, Compare 함수 객체) 방법이 있다.

## 1. 사용자 정의 타입에 operator < 를 정의

\*  $B < A == \text{false} \&\& A < B == \text{false}$  를 통해 A,B의 equal을 검사할 수 있으므로 operator == 없이 operator < 만 구현해도 됨.

\* set은 내부적으로 정렬 시에 const 빙봉자를 사용하므로 operator < 는 반드시 const reference를 인자로 받는 const 함수여야 한다.

\* operator < 는 strict weak ordering을 만족해야 하며 해당하는 조건은 다음과 (A != B 일 때) 같다. (상식적으로 operator < 를 정의하였다면 자연스레 만족됨)

\*  $A < A == \text{false}$

\*  $A < B != B < A$

\* 하나라도 만족하지 않으면

set이 정상적으로 작동하지 않는다.

\*  $(A < B \&\& B < C) == \text{true}$  면  $A < C == \text{true}$

\*  $A == B$  이면  $(A < B \|\| B < A) == \text{false}$

\*  $(A == B \&\& B == C) == \text{true}$  면  $A == C$

## 2. Compare 함수 객체를 set의 두 번째 템플릿 인자로 전달.

\* bool 타입을 반환하는 operator () 를 정의해야 하며, operator ()은 const reference 두개를 인자로 받는 const 함수이다. 내부 구현은 operator < 를 동일하다.

\* set의 두 번째 템플릿은 std::less<key> 를 인자로 받는 디폴트 인자인데, 이는 그냥 key의 operator < 를 사용하겠다는 의미이다.

## ✓ <map> : 값을 key와 value 쌍으로 저장. 템플릿 인자는 key의 타입과 Value의 타입

→ insert의 인자로 std::pair<K,V>를 받으며, std::make\_pair 함수를 통해 타입을 지정하지 않고 간단하게 std::pair 객체를 만들 수 있다.

→ insert를 사용하지 않고, operator [](key, Value)를 통해 바로 값을 추가 / 변경 가능하다. ex) m[key] = value;

→ map::iterator → first를 통해 Key에, map::iterator → second를 통해 Value에 접근할 수 있다. map::Iterator 또는 BidirectionalIterator 타입이ch.

→ operator [](key)를 통해 Value에 접근할 수 있으나, operator [](key)는 해당 pair가 map에 존재하지 않는 경우 Value에 디폴트 값을 넣어 map에 추가 후 값을 반환하므로 find를 이용해 정색 후 접근하는 것이 좋다.

## ✓ multiset(<set>), multimap(<map>) : key의 중복을 허용하는 set, map.

→ multimap의 경우 map과 달리 하나의 key에 여러 Value가 대응될 수 있기 때문에 operator []를 사용할 수 있다.

→ 또한 multimap에서 find하면서 어떤 value를 반환해야 하는지 C++ 표준에서 정해놓지 않았기 때문에 라이브러리에 따라 다른 값이 나올 수 있으므로 함부로 사용해서는 안된다.

→ 이 문제를 위해 multimap은 std::pair를 반환하는 equal\_range(key)를 제공하며, pair::first에는 해당 key의 첫 Value의 iterator, pair::second에는 해당 key의 마지막 Value의 다음 위치(end())와 유사하게 iterator가 저장되어 있다.

## ✓ <unordered\_set>, <unordered\_map> : 정렬되지 않은 set, map. ⇒ Hash!

→ Hash를 사용하기 때문에 탐색 / 삽입 / 삭제가 모두 O(1)로 작동한다.

• C++11에서 추가된 container이고, 이미 hashset, hashmap이라는 이름이 많이 사용되고 있어 이런 이름을 사용한다.

→ Hash의 특성상, 일반적인 경우 O(1)에 작동하는 rehash 및 해시의 영향으로 최악의 경우 O(n) 시간이 소요된다.

• 때문에 일반적인 경우 안전한 set과 map을 사용하고, 최적화가 매우 필요한 경우에만 Hash를 잘 설계하여 사용하는 것이 좋다.

• 물론 std::string과 C++ 기본 타입들은 Hash가 내장되어 있기 때문에 그냥 사용해도 된다.

→ 사용자 정의 타입을 unordered\_set, unordered\_map에 사용하기 위해서는 다음 2가지 (operator ==, std::hash의 템플릿 특수화)를 모두 정의해야 한다.

• 원소 간 순서가 없기 때문에 operator <>는 필요하지 않으며, set의 경우와 유사하게 const reference를 인자로 받는 operator ==을 const로 정의한다.

• std에서 제공하는 hash<type>을 특수화 한다. 이 때 특수화된 템플릿은 namespace std 안에 정의되어야 한다.

• std::hash<>는 std::size\_t (int와 동일한 타입)을 반환하는 함수 객체이다.

→ Hash 함수의 성능이 좋지 못하면 set, map보다 훨씬 못나간다며, 악의적 사용자에 의해 성능 저하가 발생할 여지도 있으므로, Hash 함수의 성능을 검증할 수 있다면 set, map을 쓰는 것이 좋다.

✓ 어떤 Associative Container를 사용해야 하는가?

→ 데이터의 존재 유무만 궁금한 경우 : set (중복을 허용하는 경우엔 multiset)

→ 데이터(key)에 대응되는 데이터(value)를 저장해야 하는 경우 : map (Key의 중복을 허용하는 경우엔 multimap)

→ 속도가 매우매우 중요하여 최적화를 해야하는 경우 : unordered\_set, unordered\_map

- Iterator : 각 컨테이너에 정의되어 있으며, STL의 Algorithm 들은 iterator 를 인자로 받아 수행함.

✓ Iterator 는 실제 포인터는 아니지만 '위치'를 통해 가리키고 있는 값에 접근할 수 있으며 그 동작도 포인터와 동일하다.

✓ 컨테이너에 원소를 추가하거나 제거하거나 되면 기존에 사용중이던 모든 iterator 들을 사용할 수 없게 된다. (linked-list로 구현된 std::list 예외)

✓ 시작과 끝이 정의된 배열 및 컨테이너에는 범위기반 for 문을 사용할 수 있다. (사용자 정의 타입도 begin() 과 end() 및 증가 연산자(++) 가 정의되어 있다면 사용 가능하다)