STL2
Sequence Container

컴퓨터학부
20152399 유제환

목차

Standard Template Library

Standard Template Library 개요

- 리스트, 큐, 스택과 같이 널리 사용되는 자료구조와 알고리즘을 구현한 라이브러리

- C++에서 사용하는 일반화 프로그래밍(Generic Programming)의 표준 라이브러리

Template을 이용한 일반화 프로그래밍

- C++은 **Template** 문법을 사용하여 일반화 프로그래밍을 구현하고 있다.
- 임의의 타입에 사용할 수 있는 자료 구조를 만들 수 있다.
- 임의의 자료 구조에 적용할 수 있는 일반화된 알고리즘을 만들 수 있다.

STL 컴포넌트(components)

- **컨테이너(Containers)**

특정 타입의 다른 객체들의 컬렉션(Collections)을 저장한다. array, deque, list, queue, stack와 같이 자료 구조에 해당한다.

- **알고리즘(Alogorithms)**

컨테이너의 내용을 초기화, 정렬, 검색 및 변환하는 방법을 제공한다.

- **이터레이터(Iterators)**

컨테이너의 요소를 단계별로 처리하는데 사용한다.

- **함수객체(Function Object, Functor)**

함수 또는 함수 포인터처럼 다뤄질 수 있는 객체이다. (C++11, functional)

Container

- C++의 클래스 템플릿으로 구현하여 타입의 유연성이 있다.
- 컴파일 시간에 자동으로 타입이 추론되어 해당 타입에 맞는 컨테이너가 구체화 된다. (코드의 효율성이 높아진다.)

```
template <class T, class Container = deque<T> > class stack;
```

```
using namespace std;

struct sample {
    int a;
    int b;
};

int main(void)
{
    stack<int> a;    // int 자료형 스택
    stack<char> b;   // char 자료형 스택
    stack<sample, list<sample>> c;  // list 컨테이너를 사용하는 sample 자료형 스택
}
```

Container 종류

이번 주제에서 다룰 범위, 선형구조

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Container - Iterator

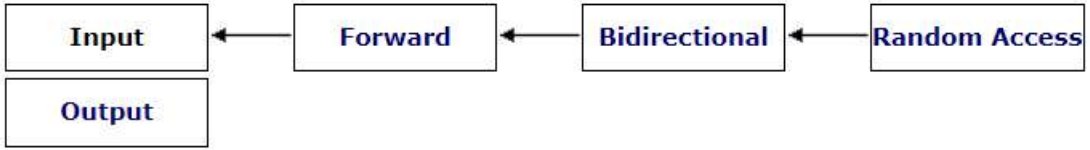
- iterator를 사용해 원소에 접근, 수정 할 수 있다.
- iterator는 컨테이너의 처리방식에 따라 달라진다.

```
int main(void)
{
    vector<int> a = { 1, 2, 3, 4 };
    for (vector<int>::iterator it = a.begin(); it != a.end(); it++)
        cout << *it << endl;
}
```

시작과 끝의 정보를 컨테이너가 가지고 있다.

포인터처럼 사용한다

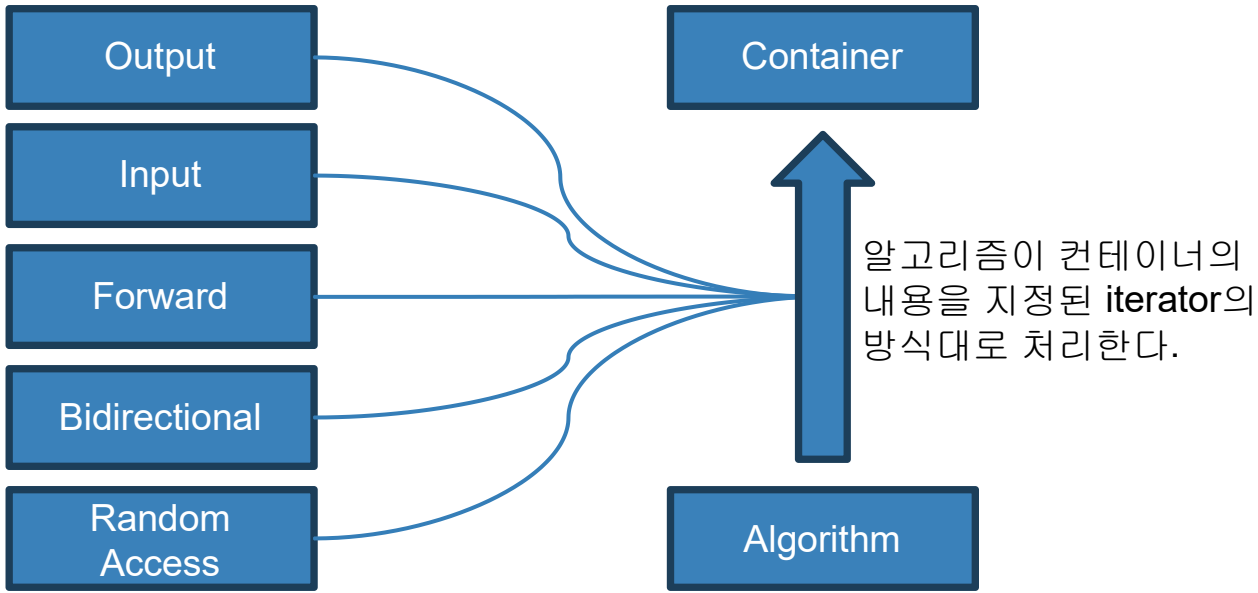
iterator category



반복자 종류	사용 방식	읽기	접근	쓰기	증감	비교
입력 반복자 (input iterator)	istream_iterator	=*p	->		++	== !=
출력 반복자 (output iterator)	ostream_iterator inserter front_inserter back_inserter			*p=	++	
순방향 반복자 (forward iterator)		=*p	->	*p=	++	== !=
양방향 반복자 (bidirectional iterator)	list set 과 multiset map 과 multimap	=*p	->	*p=	++ --	== !=
임의접근 반복자 (random access iterator)	일반 포인터 vector deque	=*p	-> []	*p=	++ -- + - += -=	== != < > <= >=

포인터가 아니라 왜 Iterator 인가?

포인터의 개념을 5가지로 일반화



결론 : 일반화 프로그래밍을 위해 사용

- STL의 Algorithm은 자신이 처리할 수 있는 iterator 유형에 대해서 컨테이너에 관계 없이 연산을 수행한다.
- Algorithm을 딱 하나 정의함으로서 연산자(operator)의 정의가 넘쳐나는 것을 방지할 수 있다.

```
int main(void)
{
    vector<int> a = { 10, 20, 30, 30, 20, 10, 10, 20 };
    cout << count(a.begin(), a.end(), 20) << endl;
}
```

Random Access Iterator를 사용

Random Access Iterator는 Input Iterator에 유효하다.

count 함수는 input iterator를 받는다.

```
template <class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
    count (InputIterator first, InputIterator last, const T& val);
```

count라는 알고리즘은 컨테이너의 자료구조가 무엇이든 간에 input interator에 유효하면 알고리즘이 수행된다.

Sequence Container

array (C++11)

- 길이가 고정되어 있는 정적배열 (static array)
- C에서 int a[10]; 이런식으로 선언했던 정적 배열과 유사
- 정적배열에 STL을 접목시켰다.
- random access iterator(임의 접근 반복자)를 사용한다.

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```

(since C++11)

```
int main(void)
{
    array<int, 3> a; // default initialization
    array<int, 5> b = { 1, 2, 3 }; // initializer-list
    array<int, 4> c{ 1, 2, 3, 4 }; // uniform initialization
    array<int> d; // error! 길이가 반드시 필요함
}
```

<error-type> d
error! 길이가 반드시 필요함

클래스 템플릿 "std::array"에 대한 인수가 너무 적습니다.

array Initialization

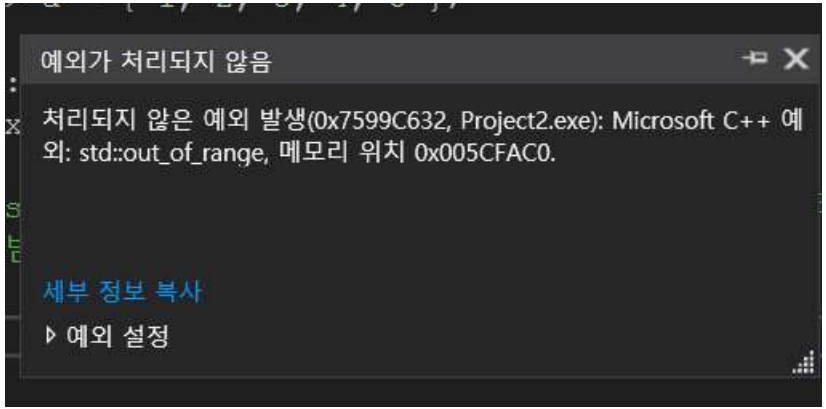
Example

```
// constructing arrays
#include <iostream>
#include <array>

// default initialization (non-local = static storage):
std::array<int,3> global;                 // zero-initialized: {0,0,0}

int main ()
{
  // default initialization (local = automatic storage):
  std::array<int,3> first;                // uninitialized:    {?,?,?}

  // initializer-list initializations:
  std::array<int,3> second = {10,20};     // initialized as:   {10,20,0}
  std::array<int,3> third = {1,2,3};      // initialized as:   {1,2,3}

  // copy initialization:
  std::array<int,3> fourth = third;       // copy:             {1,2,3}

  std::cout << "The contents of fourth are:";
  for (auto x:fourth) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

array 왜 만들었을까?

- 안전한 접근

```
a.at(5); // segmentaion fault가 아닌, std::out_of_range라는 정확한 예외를 알려줌!
         // 범위에 대한 유효성 검사를 하기 때문.
```



array 왜 만들었을까?

- 범위 기반 루프 (range based loop)

```
for (auto x : a)
    cout << x << endl;

for (auto it = a.begin(); it != a.end(); it++)
    cout << *it << endl;
```

개발자가 배열의 사이즈를 몰라도 안전하게 배열을 돌릴 수 있다.

array 왜 만들었을까?

- 배열을 함수 인자로 넘길 때 포인터를 사용하지 않음

```
array<int, 5> b = { 1, 2, 3, 4, 5 };
foo(b);
```

함수에서 포인터 타입을 사용하지 않고 객체 타입 그대로 사용한다. 그래서 배열의 사이즈를 함수에 별도로 넘겨줄 필요가 없다.

vector

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;                                                    (1)

namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;   (2)  (since C++17)
}
```

- 길이가 가변적인 배열 (dynamic array)
- 동적할당... 듣기만 해도 거북하고 하기 싫다. 하지만! 벡터가 있다면 걱정 없다.
- 삽입시 새로운 공간이 필요하면 알아서 재할당을 해준다.
- resize(), reserve()를 이용해 명시적으로 공간을 늘릴 수 있다.
- random access iterator(랜덤 접근 반복자)를 사용한다.
- 다른 시퀀스 컨테이너에 비해 마지막 위치에서 삽입, 삭제가 가장 빠르다.
- 배열이기 때문에 중간 삽입 삭제가 용이하지 않다.

vector의 메모리 사용

- 배열의 길이를 증가시키고 싶을 때 마다 재할당(reallocation)을 하게 되면 엄청나게 느리다.
- 이를 개선하기 위해 벡터 원소의 개수 보다 훨씬 더 큰 메모리를 미리 잡는다.
- 따라서 정적 배열보다 메모리를 훨씬 많이 잡아먹는다.

vector의 capacity

```
int main(void)
{
    vector<int> v;

    for (int i = 0; i < 100; i++)
        v.push_back(i);

    cout << v.size() << endl;
    cout << v.max_size() << endl;
    cout << v.capacity() << endl;
}
```

```
100
1073741823
141
```

size는 생성된 원소 개수
max_size는 생산할 수 있는 최대 개수
capacity는 할당된 메모리 공간의 개수

size <= capacity <= max_size

*Vector **resize()** vs reserve()*

```
int main(void)
{
    vector<int> v;

    for (int i = 0; i < 100; i++)
        v.push_back(i);

    v.resize(10);

    for (auto x : v)
        cout << x << " ";
    cout << endl;

    cout << "size: " << v.size() << endl;
    cout << "capacity: " << v.capacity() << endl;

    v.resize(15);

    for (auto x : v)
        cout << x << " ";
    cout << endl;

    cout << "size: " << v.size() << endl;
    cout << "capacity: " << v.capacity() << endl;

}
```

```
0 1 2 3 4 5 6 7 8 9
size: 10
capacity: 141
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0
size: 15
capacity: 141
```

*vector resize() vs **reserve()***

```
int main(void)
{
    vector<int> v;

    for (int i = 0; i < 100; i++)
        v.push_back(i);

    v.reserve(10);

    for (auto x : v)
        cout << x << " ";
    cout << endl;

    cout << "size: " << v.size() << endl;
    cout << "capacity: " << v.capacity() << endl;

    v.reserve(200);

    for (auto x : v)
        cout << x << " ";
    cout << endl;

    cout << "size: " << v.size() << endl;
    cout << "capacity: " << v.capacity() << endl;

}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
size: 100
capacity: 141
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
size: 100
capacity: 200
```

vector resize() vs reserve() 결론

resize()는 size를 reserve()는 capacity를 증가시킨다.

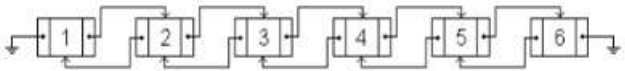
capacity가 증가하면 새로운 메모리가 할당(allocation)된 것이고, size가 증가하면 새로운 메모리가 할당됨과 동시에 원소가 생성(construction)된 것이다.

list

std::list

Defined in header <list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;                                                    (1)

namespace pmr {
    template <class T>
    using list = std::list<T, std::pmr::polymorphic_allocator<T>>;   (2)  (since C++17)
}
```



- 더블 링크드 리스트로 구현됐다.
- 데이터를 저장할 때 연속적인 메모리를 사용하지 않는다.
- 다른 시퀀스 컨테이너와 비교했을 때, 삽입, 삭제가 용이하다.
- vector와 달리 삽입을 위해 미리 공간을 할당해두지 않고, 삽입할 때 공간을 새로 할당받아 생성한다.
- bidirectional iterator(양방향 반복자)를 사용한다.

vector vs list 삽입 속도 비교

1000만개의 데이터를 마지막 위치에 삽입할 때 (동일하게 push_back)

```
vector push_back() Runtime: 9:744364(sec:usec)
list push_back() Runtime: 14:495184(sec:usec)
```

10만개의 중간 위치에 삽입할 때 (미리 10만의 사이즈를 할당함)

```
for (int i = 0; i < 100000; i++) {
    v.insert(v.begin() + 50000, i);
}
```

```
vector insert() Runtime: 2:967859(sec:usec)
list insert() Runtime: 0:289833(sec:usec)
```

```
it = li.begin();
for (int i = 0; i < 50000; i++, it++);

for (int i = 0; i < 100000; i++) {
    li.insert(it, i);
}
```

결론 : 삽입하는 과정에서 둘의 장단점이 확연하게 드러난다.

vector vs list 정렬 속도 비교

100만개의 데이터를 정렬할 때 비교

```
li.sort();
```

```
sort(v.begin(), v.end());
```

```
alogrithm sort() vector Runtime: 2:074807(sec:usec)
list sort() Runtime: 60:426576(sec:usec)
```

random access가 가능하면 algorithm 헤더의 sort()를 쓸 수 있다.
하지만 List는 random access가 불가능하기 때문에 sort() 알고리즘을 사용할 수 없다.
대신에 list는 멤버 함수로 sort()를 제공한다. (속도가 매우 느림)

결론 : 정렬을 할꺼면 vector와 같이 random access를 지원하는 자료구조를 쓰자!

forward_list (C++11)

std::forward_list

Defined in header <forward_list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class forward_list;
```

(1) (since C++11)

```
namespace pmr {
    template <class T>
    using forward_list = std::forward_list<T, std::pmr::polymorphic_allocator<T>>;
}
```

(2) (since C++17)

- list가 더블 링크드 리스트였다면, forward_list는 싱글 링크드 리스트이다.
- 싱글 링크드 리스트를 사용해 forward_list 보다 메모리 측면에서 우수하다. list의 성능 개선을 목적으로 개발 되었다.
- forward iterator(순방향 반복자)를 사용한다.

list vs forward_list

- **iterator의 차이 때문에 삽입 / 삭제 방식이 다르다.**

싱글 링크드 리스트라 삽입과 삭제가 모두 어떤 위치에 다음에서 이루어질 수 밖에 없다. (next 노드밖에 모르기 때문)

따라서 insert, erase가 없고 insert_after, erase_after가 구현되어 있다.

- **size()**

forward_list는 구현당시 list의 성능을 개선하고 메모리 오버헤드를 최소화 하려는 목표가 있었다. 그래서 size를 카운트 해주는 멤버를 뺐고, 그러면서 함수 멤버 size()도 뺐다고 한다.

(사이즈를 알고 싶다면 algorithm의 distance()를 사용해야한다.)

deque

std::deque

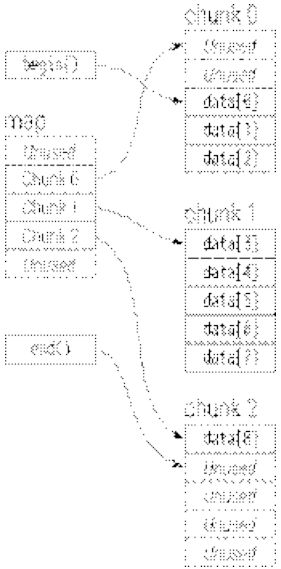
Defined in header <deque>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;                                                          (1)

namespace pmr {
    template <class T>
    using deque = std::deque<T, std::pmr::polymorphic_allocator<T>>;    (2)  (since C++17)
}
```

- double ended queue의 약자.
- 양 쪽 끝에서 사이즈가 증가 할 수 있다.
- random access iterator(임의 접근 반복자)를 사용한다.
- 청크 방식을 사용해서 메모리 재할당에서 발생하는 성능저하를 개선했다.
- 모든 요소가 연속 저장되어 있다는 것을 보장하지 않는다.
- 시작과 끝에 삽입이 빈번한 경우에 사용하면 좋다.

vector vs deqeue

- 벡터는 메모리가 부족할 때 새로운 메모리 블록을 할당하고 기존에 있던 메모리 블록을 복사하는 방식이다.
- 덱은 메모리가 부족할 때 일정 크기의 새로운 메모리 블록(Chunk)를 할당하고 복사하지 않고 매핑하는 방식을 사용한다.
- 덱은 내부적으로 이러한 메모리 블록들을 매핑하고 있어 논리적으로 하나의 메모리 블록인 것처럼 동작한다.



Container Adaptor

container apdator

- 기존의 컨테이너에서 기능이 제한되거나 변형된 컨테이너를 말한다.
- container adpator는 실제 저장소가 아니라 내부 컨테이너가 따로 있다.
- 동작에 대한 인터페이스를 구현하여, 내부 컨테이너가 특정 형태의 동작만을 수행하도록 유도한다.

stack

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

- 구현된 인터페이스를 토대로 알 수 있는 점은 stack의 컨테이너로 사용되려면 empty(), size(), back(), push_back(), pop_back()을 지원해야한다.
- deque, vector, list가 이를 만족한다.
- 내부 컨테이너가 무엇이든 간에 캡슐화되어 그 성질을 잃게 된다. (stack의 컨테이너로 deque을 썼다고해서 pop_front()를 할 수는 없다.)

Class std::stack

```
template <class T, class Container = std::deque<T> >
class stack {
 public:
    typedef typename Container::value_type value_type;
    typedef typename Container::reference reference;
    typedef typename Container::const_reference const_reference;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

 protected:
    Container c;

 public:
    explicit stack(const Container&);
    explicit stack(Container&& = Container());
    template <class Alloc> explicit stack(const Alloc&);
    template <class Alloc> stack(const Container&, const Alloc&);
    template <class Alloc> stack(Container&&, const Alloc&);
    template <class Alloc> stack(const stack&, const Alloc&);
    template <class Alloc> stack(stack&&, const Alloc&);

    bool empty() const {
        return c.empty();
    }
    size_type size() const {
        return c.size();
    }
    reference top() {
        return c.back();
    }
    const_reference top() const {
        return c.back();
    }
    void push(const value_type& x) {
        c.push_back(x);
    }
    void push(value_type&& x) {
        c.push_back(std::move(x));
    }
    template <class... Args> void emplace(Args&&... args) {
        c.emplace_back(std::forward<Args>(args)...);
    }
    void pop() {
        c.pop_back();
    }
    void swap(stack& s) noexcept(noexcept(swap(c, s.c))) {
        using std::swap;
        swap(c, s.c);
    }
};
```

queue

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

- queue는 empty(), size(), front(), back(), push_back(), pop_front()를 지원해야한다.
- deque과 list 컨테이너가 이를 만족한다.

Class std::queue

```
template <class T, class Container = deque<T> >
class queue {
 public:
    typedef typename Container::value_type value_type;
    typedef typename Container::reference reference;
    typedef typename Container::const_reference const_reference;
    typedef typename Container::size_type size_type;
    typedef Container container_type;
 protected:
    Container c;
 public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());
    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(const queue&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    bool empty() const {
        return c.empty();
    }
    size_type size() const {
        return c.size();
    }
    reference front() {
        return c.front();
    }
    const_reference front() const {
        return c.front();
    }
    reference back() {
        return c.back();
    }
    const_reference back() const {
        return c.back();
    }
    void push(const value_type& x) {
        c.push_back(x);
    }
    void push(value_type&& x) {
        c.push_back(std::move(x));
    }
    template <class... Args> void emplace(Args&&... args) {
        c.emplace_back(std::forward<Args>(args)...);
    }
    void pop() {
        c.pop_front();
    }
    void swap(queue& q) noexcept(noexcept(swap(c, q.c))) {
        using std::swap;
        swap(c, q.c);
    }
};
```

priority_queue

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

- 우선순위대로 큐를 만들 수 있다. std::less<T>를 사용하면 내림차순으로, std::greater<T>를 사용하면 오름차순으로 정렬할 수 있다.
- 힙(Heap) 구조로 되어있으며, algorithms의 make_heap으로 내부의 vector 컨테이너를 힙 구조로 바꿔버린다.

Class std::priority_queue

```
template <class T,
          class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
class priority_queue {
 public:
    typedef typename Container::value_type value_type;
    typedef typename Container::reference reference;
    typedef typename Container::const_reference const_reference;
    typedef typename Container::size_type size_type;
    typedef Container container_type;
 protected:
    Container c;
    Compare comp;
 public:
    priority_queue(const Compare& x, const Container&);
    explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x, const Container&);
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x = Compare(), Container&& = Container());
    template <class Alloc> explicit priority_queue(const Alloc&);
    template <class Alloc> priority_queue(const Compare&, const Alloc&);
    template <class Alloc> priority_queue(const Compare&,
                                          const Container&, const Alloc&);
    template <class Alloc> priority_queue(const Compare&,
                                          Container&&, const Alloc&);
    template <class Alloc> priority_queue(const priority_queue&, const Alloc&);
    template <class Alloc> priority_queue(priority_queue&&, const Alloc&);

    bool empty() const {
        return c.empty();
    }
    size_type size() const {
        return c.size();
    }
    const_reference top() const {
        return c.front();
    }
    void push(const value_type& x);
    void push(value_type&& x);
    template <class... Args> void emplace(Args&&... args);
    void pop();
    void swap(priority_queue& q) noexcept( noexcept(swap(c, q.c))
                                        && noexcept(swap(comp, q.comp))) {
        using std::swap;
        swap(c, q.c);
        swap(comp, q.comp);
    }
};
```

heap 예제

Example

```
// range heap example
#include <iostream>     // std::cout
#include <algorithm>    // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

Output:

```
initial max heap   : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99
```

예제를 보면 priority_queue 내부에서 어떤 일이 일어나는지 유추해볼 수 있다.

들어주셔서 감사합니다.