C++ Programming

15th Study: Standard Template Library #1

- STL
- sequence container : vector, deque, list

C++ Korea 윤석준 (icysword77@gmail.com)

STL

Standard Template Library

STL ? 언제 ?

- Bijarne Stroustrup이 1978년부터 C++ 개발
- 1993년 Alexander Stepanov 가 제안
- 1994년 C++ 표준 위원회에서 초안이 통과

Bijarne Stroustrup

Alexander Stepanov

Standard Template Library

표준 템플릿 라이브러리

위키백과, 우리 모두의 백과사전.

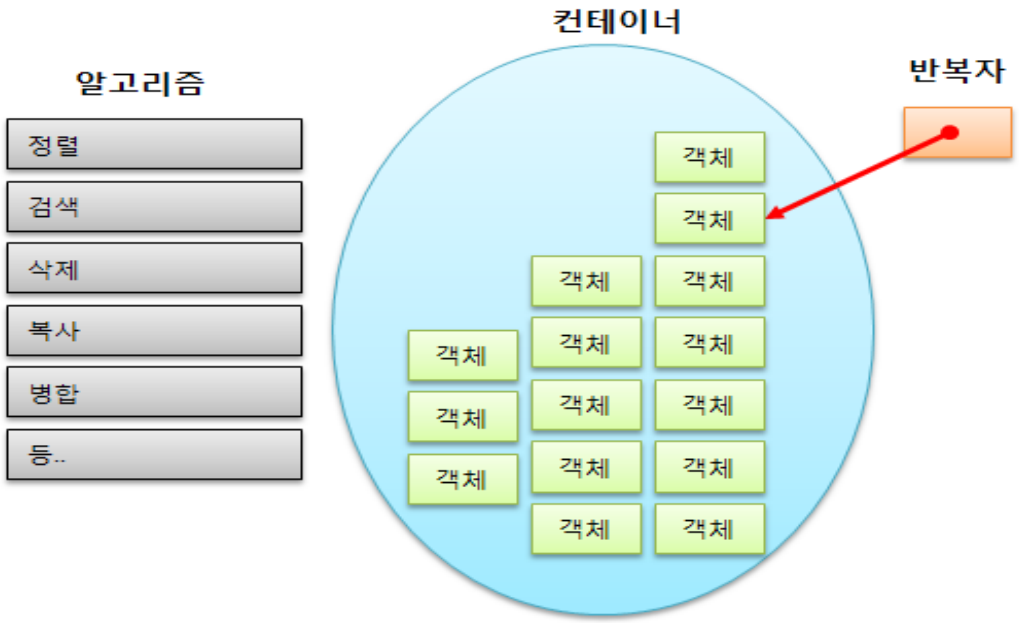
표준 템플릿 라이브러리(Standard Template Library: **STL**)는 C++에서 일반적인 자료 구조와 알고리즘을 구현해 놓은 라이브러리의 집합이다. 지원하는 자료구조에는 vector, map, set 등이 있으며, 여러 가지 탐색 변경 알고리즘을 지원해 주고 있다. vector와 같은 자료구조에 삽입할 변수의 형이 정해지 있지 않고, 일반적인 형이라고 가정한 뒤 vector와 같은 컨테이너가 구현되어 있다. 즉 이 때는 C++언어의 템플릿 기능을 이용하고 있다. 이처럼 자료의 유형에 상관없이 구현되어 있기 때문에 generic이라고 말하기도 한다.

http://ko.wikipedia.org/wiki/표준_템플릿_라이브러리

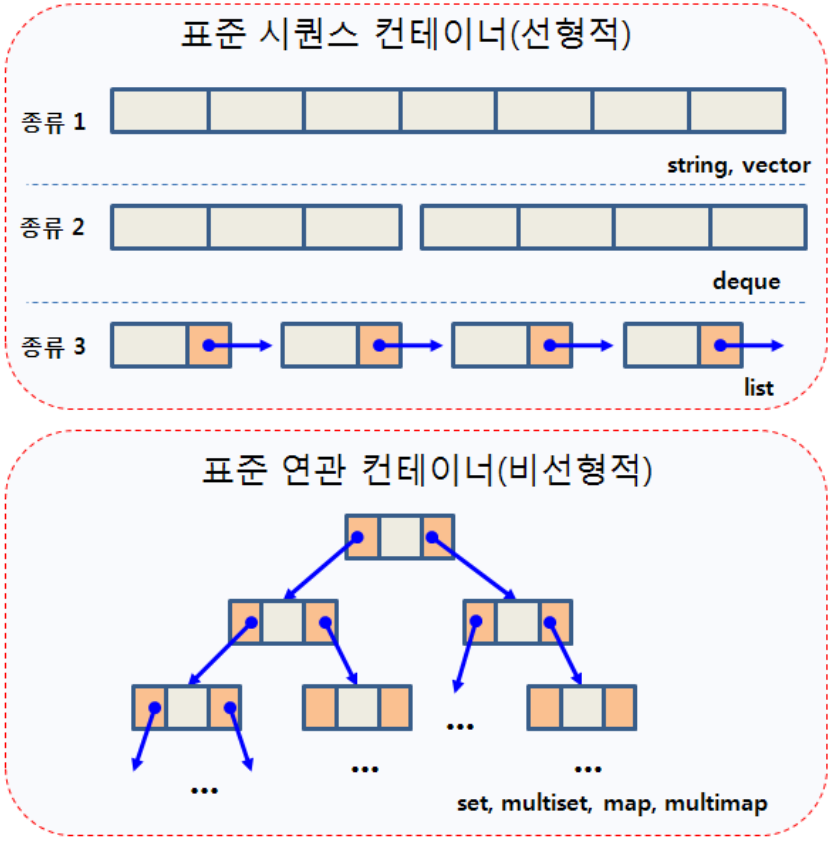
STL의 구성요소

- 컨테이너 (container) : 객체를 저장
- 반복자 (iterator) : 컨테이너에 저장된 객체에 대한 포인터
- 알고리즘 (algorithm) : 정렬, 삭제, 검색 등에 대한 함수
- 함수 객체 (function object) : 함수처럼 동작하는 객체
- 어댑터 (adaptor) : 구성 요소를 새로운 구성요소로 보이게 하는 기능
- 할당기 (allocator) : 컨테이너의 메모리 할당 정책을 담당

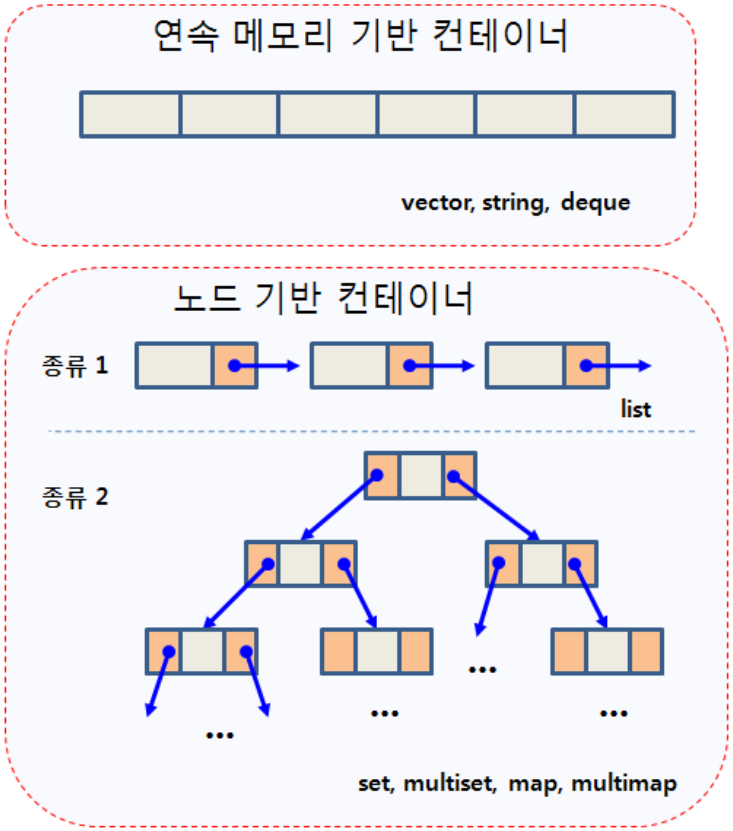
STL의 구성요소



컨테이너 분류 (데이터 구조)



컨테이너 분류 (메모리)


연속 메모리 기반 컨테이너
vector, string, deque
노드 기반 컨테이너
종류 1
list
종류 2
set, multiset, map, multimap

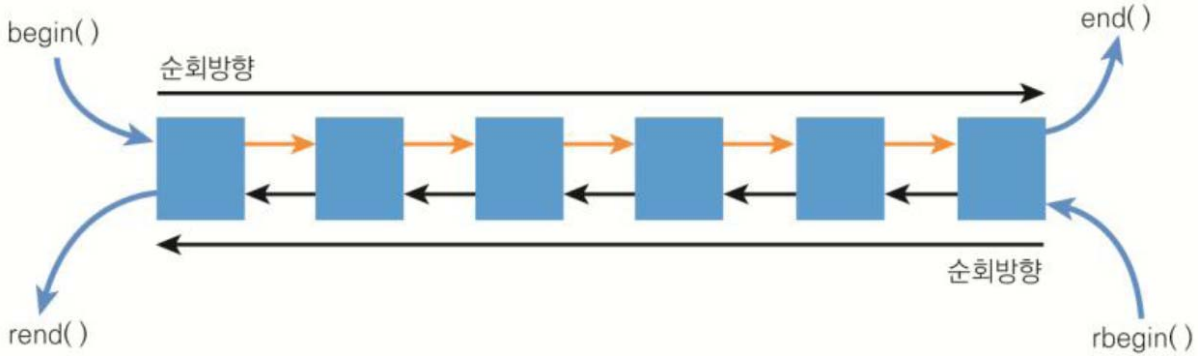
iterator

반복자

std::iterator

- STL의 container에 저장된 데이터에 접근하는 방법
- STL의 컨테이너<자료형>::iterator 변수이름
- 포인터 연산이 대부분 그대로 사용

  ++, -- 로 이동, -> 로 내부함수 접근, * 로 개체에 직접 접근



Sequence container

선형적 저장공간

Sequence container

- 저장 원소에 상대적인 위치 (순서) 개념이 있는 저장공간
- `std::vector` : 가장 많이 사용됨.
  (사실 이것만 알아도...)
- `std::deque` : 앞, 뒤로 데이터 삽입, 삭제가 가능
  (하지만 나머지는 vector보다 느림)
- `std::list` : 중간에 데이터 삽입, 삭제가 가장 빠름
  (하지만 나머지는 다 느림)

vector

std::vector

std::vector

- STL에서 가장 많이 사용되는 container library
- C++ 배열과 비슷하지만, 더 강력한 도구
- 메모리 상에 연속된 공간에 저장된다.

  `memcpy()`, `memset()` 등의 함수를 이용하여 직접적인 데이터 수정이 가능

  할당된 크기 이상으로 변경 시 새로운 곳에 새로 할당 받아야 한다.
- 뒤쪽에서만 데이터 삽입, 삭제가 가능하다.
- 별도 함수 이용으로 중간에 삽입, 삭제도 가능하나, 이 경우 많은 데이터 이동이 발생한다.

배열 과 std::vector의 특징

배열

- 크기가 고정적이다.
  (단점)
- 중간에 데이터 삽입, 삭제가 힘들다.
  (단점)
- 구현이 쉽다.
  (장점)
- 랜덤 접근이 가능하다.
  (장점)

std::vector

- 크기가 동적으로 변한다.
  (장점)
- 중간에 데이터 삽입, 삭제가 가능하다.
  (장점)
- 구현이 쉽다.
  (장점)
- 랜덤 접근이 가능하다.
  (장점)

언제 std::vector를 사용 ?

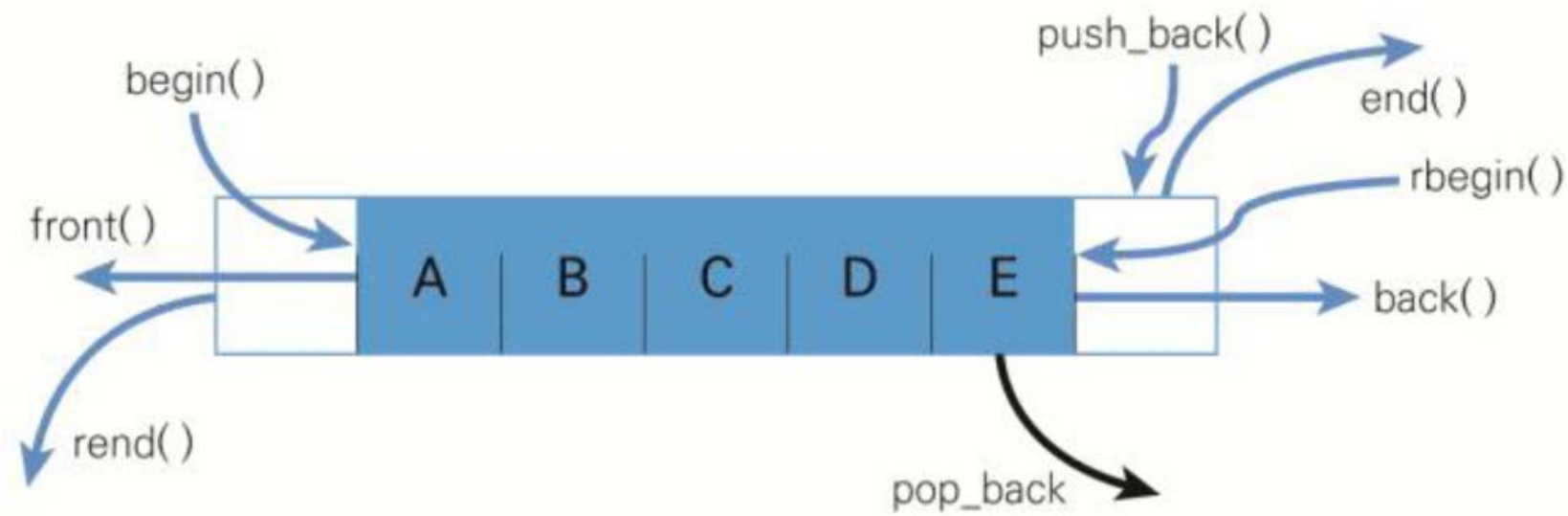
- 미리 크기를 알 수 있는 데이터 저장에 가장 적합
- 크기가 가변적인 경우일지라도,

  뒤쪽으로만 데이터 삽입, 삭제가 이루어지는 경우

  앞에서도 삽입, 삭제가 필요하다면 std::deque가 더 좋음

  중간에 데이터 삽입, 삭제가 자주 일어난다면 std::list가 더 좋음
- 랜덤 접근이 자주 발생하는 경우
- 빠른 속도를 요구하는 경우

std::vector의 기본 함수



std::vector 사용 방법 #1

```
#include <vector>          // 해더 파일

std::vector<int> vec;      // 선언

vec.push back(10);         // 추가
vec.pop_back();            // 삭제

vec.resize(10, 0);         // 초기화 : 10의 크기를 0으로 채움
vec[3] = 10;               // 랜덤 접근

std::cout << vec[3] << std::endl; // 랜덤 접근
```

std::vector 사용 방법 #2

```
int nSum = 0;
for (int i = 0; i < vec.size(); i++)
{
    nSum += vec[i];
}

// auto = std::iterator<std::vector<int>>
for (auto it = vec.begin(); it != vec.end(); it++)
{
    nSum += (*it);
}
```

deque

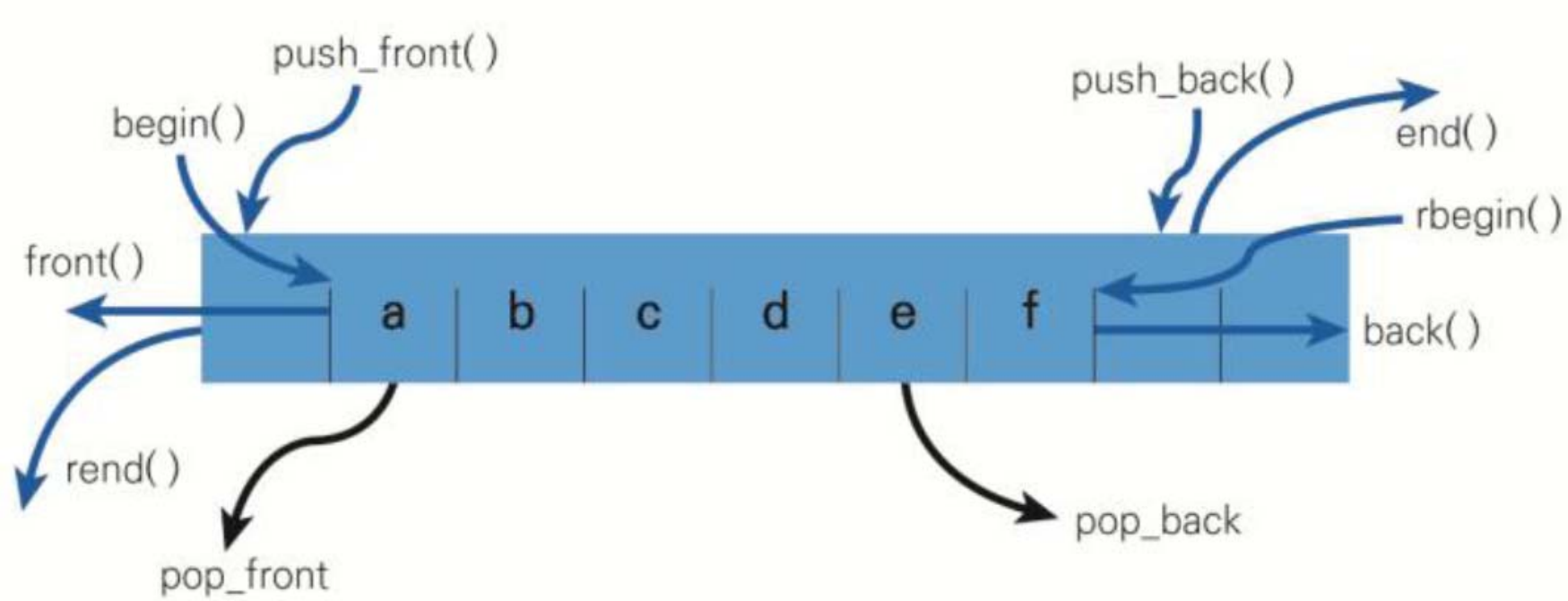
std::deque

std::deque

- 앞, 뒤 모두에서 데이터 삽입, 삭제가 가능
- 하지만 다른 기능들은 `std::vector`보다 성능이 떨어짐
- 웬만해서는 안 씀

	deque	vector
크기 변경 가능	O	O
앞에 삽입, 삭제 용이	O	X
뒤에 삽입, 삭제 용이	O	O
중간 삽입, 삭제 용이	X	X
순차 접근 가능	O	O

std::deque의 기본 함수



std::deque 사용 방법 #1

```
#include <deque>          // 헤더 파일

std::deque<int> deq;      // 선언

deq.push_back(10);         // 뒤에 추가
deq.pop_back();            // 뒤에 삭제
deq.push_front(20);        // 앞으로 추가
deq.pop_front();           // 앞에 삭제

deq.resize(10, 0);         // 초기화 : 10의 크기를 0으로 채움
deq[3] = 10;               // 랜덤 접근

std::cout << deq[3] << std::endl; // 랜덤 접근
```

std::deque 사용 방법 #2

```
int nSum = 0;
for (int i = 0; i < deq.size(); i++)
{
    nSum += vec[i];
}

// auto = std::vector<int>::iterator
for (auto it = deq.begin(); it != deq.end(); it++)
{
    nSum += (*it);
}
```

list

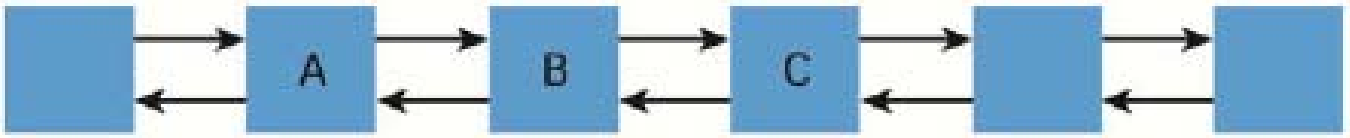
std::list

std::list

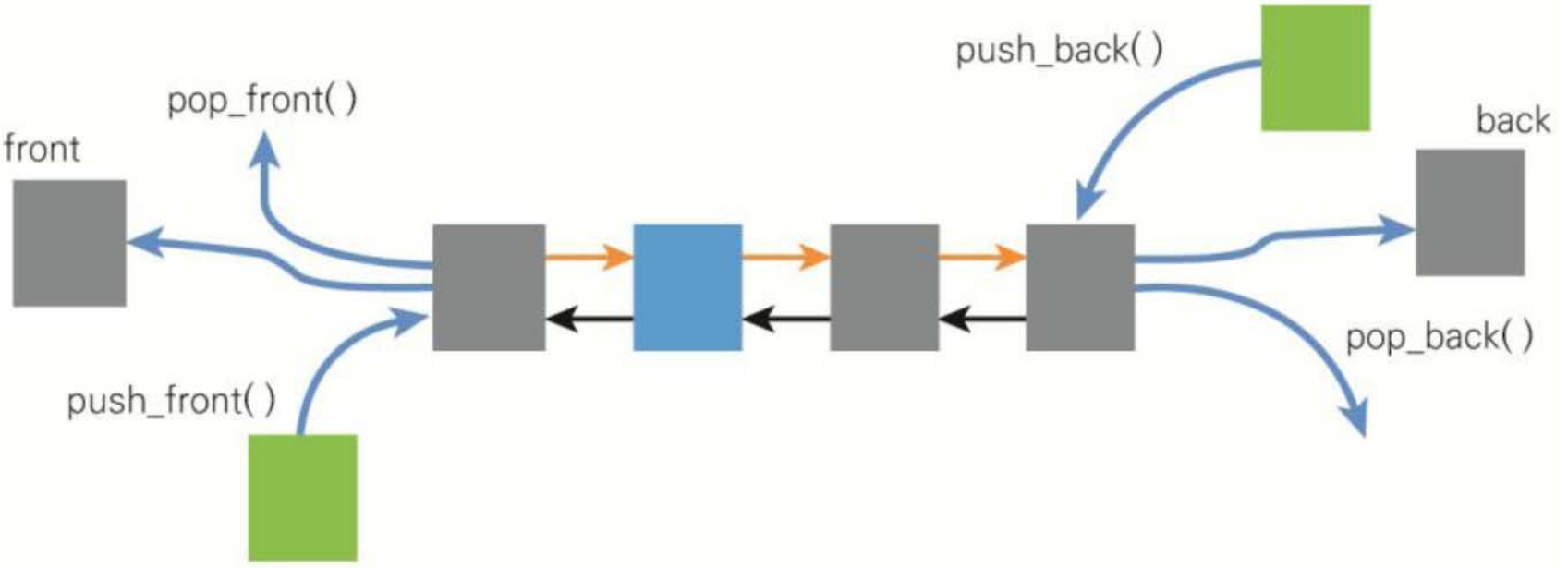
- 자료구조의 Linked List를 template로 구현해 놓은 것
- 중간에 데이터 삽입, 삭제가 가장 빠름

  그것 빼곤 가장 불편함
- 웬만해서 쓸 일이 잘 없음

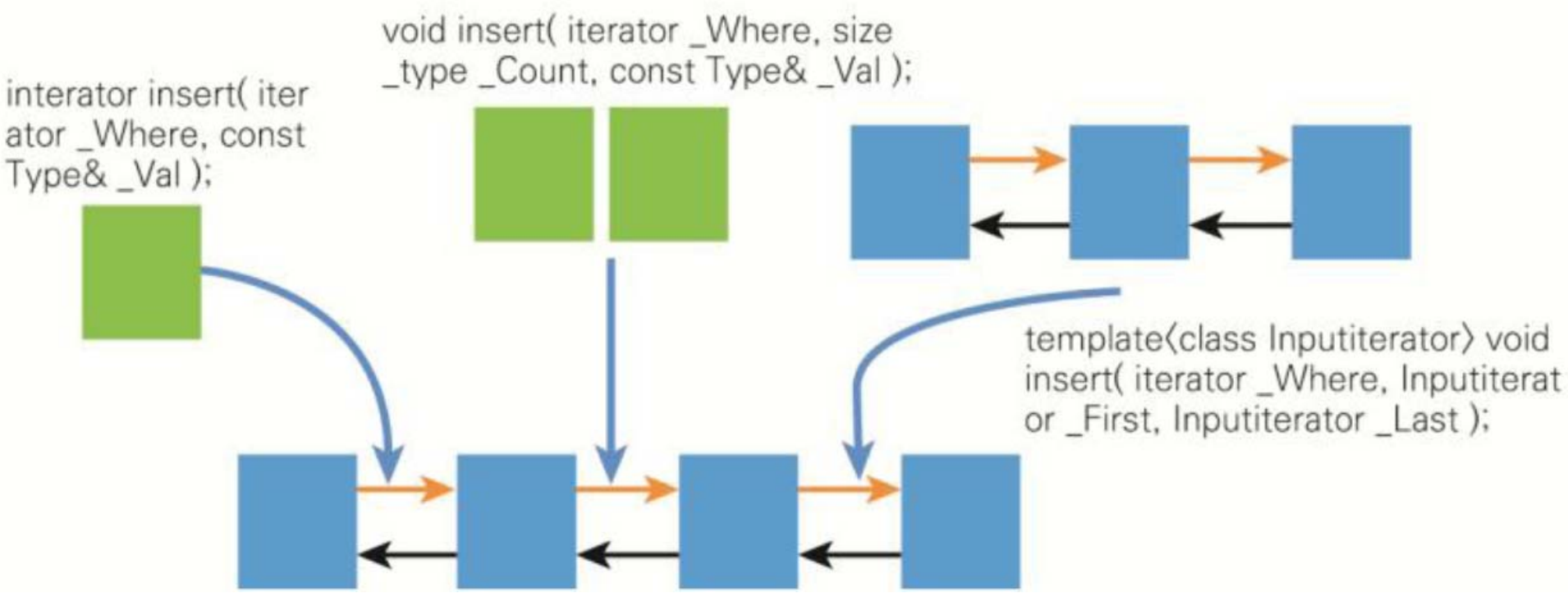
  난 한번도 없음



std::list의 앞,뒤 삽입



std::list의 insert (std::vector, std::deque도 같음)



std::list 예제 #1

```
#include <list>

std::list<int> lst;

lst.push_back(5);              // 5
lst.push_back(10);             // 5 -> 10
lst.push_front(1);             // 1 -> 5 -> 10

auto it = lst.begin();         // 첫번째 위치, it -> 1 -> 5 -> 10
it++;                          // 두번째 위치, 1 -> it -> 5 -> 10
auto at = lst.insert(it, 2); // 2 삽입, 1 -> at -> 2 -> it -> 5 -> 10
lst.insert(at, 3);             // 1 -> 3 -> at -> 2 -> it -> 5 -> 10
lst.insert(it, 4);             // 1 -> 3 -> at -> 2 -> 4 -> it -> 5 -> 10

for (auto iter = lst.begin(); iter != lst.end(); iter++)
    std::cout << (*iter) << '\t';
```

std::list 예제 #2

```
std::list<int> lstB;
lstB.push_back(20);          // 20
lstB.push_back(30);          // 20 -> 30

                             // 1 -> 3 -> at -> 2 -> 4 -> it -> 5 -> 10
lst.insert(it, lstB.begin(), lstB.end());
                          // 1 -> 3 -> at -> 2 -> 4 -> 20 -> 30 -> it -> 5 -> 10

for (auto iter = lst.begin(); iter != lst.end(); iter++)
    std::cout << (*iter) << '\t';
std::cout << std::endl;
```