

# 게임 자료구조와 알고리즘

-CHAPTER16-

SOULSEEK

# 목차

1. **STL**의 이해
2. **vector** - 시퀀스 컨테이너

# **STL**의 이해

# 1. STL의 이해

- **Standard Template Library**
- 프로그램에 필요한 자료구조와 알고리즘을 템플릿으로 제공하는 라이브러리
- 자료구조와 알고리즘은 서로 반복자라는 구성 요소를 통해 연결한다
- 효율성, 일반화 프로그램(재활용성), 확장성이 특징이다.

## 구성 요소

### 컨테이너(Container)

- 객체를 저장하는 객체로 컬렉션 혹은 자료구조라고도 한다

### 반복자(Iterator)

- 포인터와 비슷한 개념으로 컨테이너의 원소를 가리키고, 가리키는 원소에 접근하여 다음 원소를 가리키게 하는 기능을 한다

### 알고리즘(Algorithm)

- 정렬, 삭제, 검색, 연산 등을 해결하는 일반화된 방법을 제공하는 함수 템플릿이다

### 함수 객체(Function Object)

- 함수처럼 동작하는 객체로 **operator()** 연산자를 오버로딩한 객체, 컨테이너와 알고리즘 등에 클라이언트 정책을 반영한다

### 어댑터(Adapter)

- 구성요소의 인터페이스를 변경해 새로운 인터페이스를 갖는 구성 요소로 변경한다

### 할당기(Allocator)

- 컨테이너의 메모리 할당 정책을 캡슐화한 클래스 객체로 모든 컨테이너는 자신만의 기본 할당기를 가지고 있다

# 1. STL의 이해

함수 객체

|         |
|---------|
| less    |
| greater |
| ...     |

알고리즘



컨테이너

|    |    |
|----|----|
| 객체 | 객체 |

반복자

# 1. STL의 이해

## 컨테이너

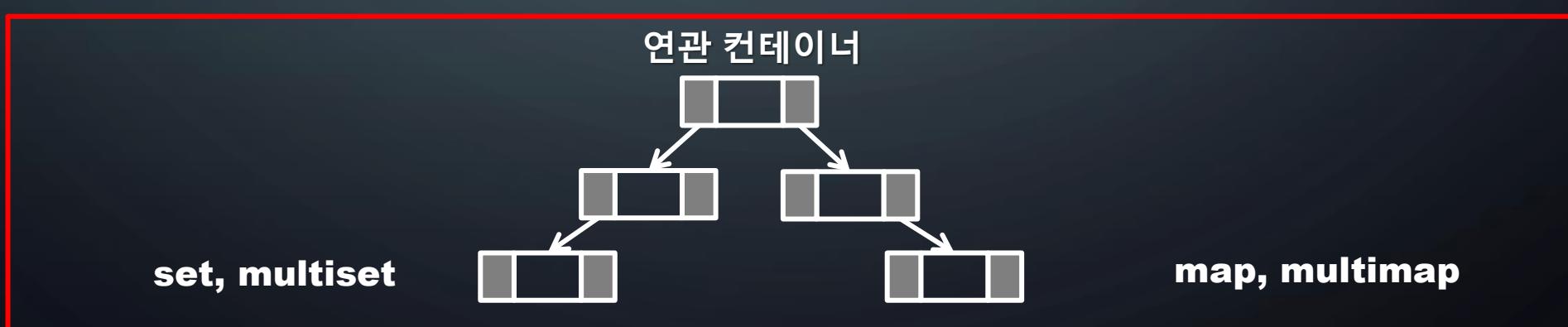
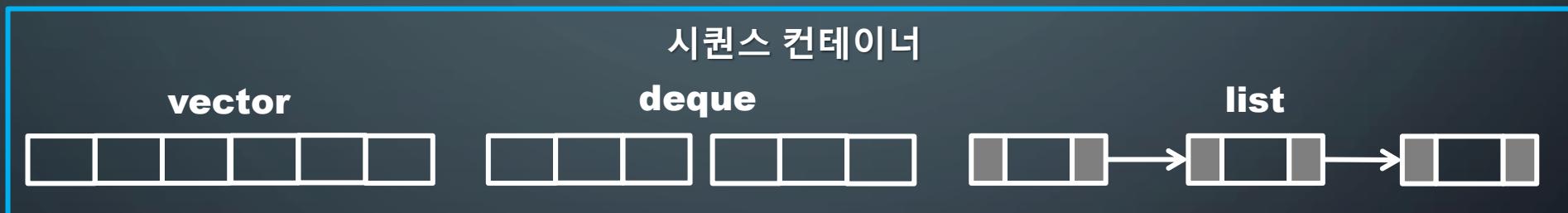
- 같은 타입을 저장, 관리할 목적으로 만들어진 클래스.
- 시퀀스 컨테이너, 연관 컨테이너 두 가지가 있다.
- 데이터 저장 방식에 따라 배열 기반, 노드 기반으로 나뉜다.

### 시퀀스 컨테이너(sequence container)

- 컨테이너 원소가 자신만의 삽입위치(순서)를 가지는 컨테이너 – **vector, deque, list** : 선형적

### 연관 컨테이너(associative container)

- 저장 원소가 삽입 순서와 다르게 특정 기준에 의해 자동 정렬되는 컨테이너 – **set, multiset, map, multimap** : 비선형



# 1. STL의 이해

## 배열 기반 컨테이너(array based container)

- 데이터 여러 개가 하나의 메모리 단위에 저장된다.(vector, deque)

## 노드 기반 컨테이너(node based container)

- 데이터 하나를 하나의 메모리 단위에 저장한다.

### 노드 기반 컨테이너

#### vector

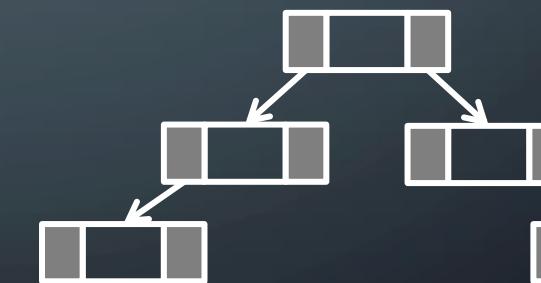


#### deque



### 배열 기반 컨테이너

#### list



**set, multiset, map, multimap**

# 1. STL의 이해

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

```
Int main()
{
```

// int 타입(정수)을 저장하는 컨테이너 v를 생성한다.  
**vector<int> v;**

```
v.push_back(10);
v.push_back(20);
v.push_back(30);
v.push_back(40);
v.push_back(50);
```

```
for(unsigned int i = 0; i < v.size(); ++i)
    cout << v[1] << endl; // v[1]는 i번째 index의 정수 반환
```

```
return 0;
}
```

추가한 순서대로 출력되며 **operator[]**을 통해 저장 원소에 접근 할 수 있다.

# 1. STL의 이해

## 반복자

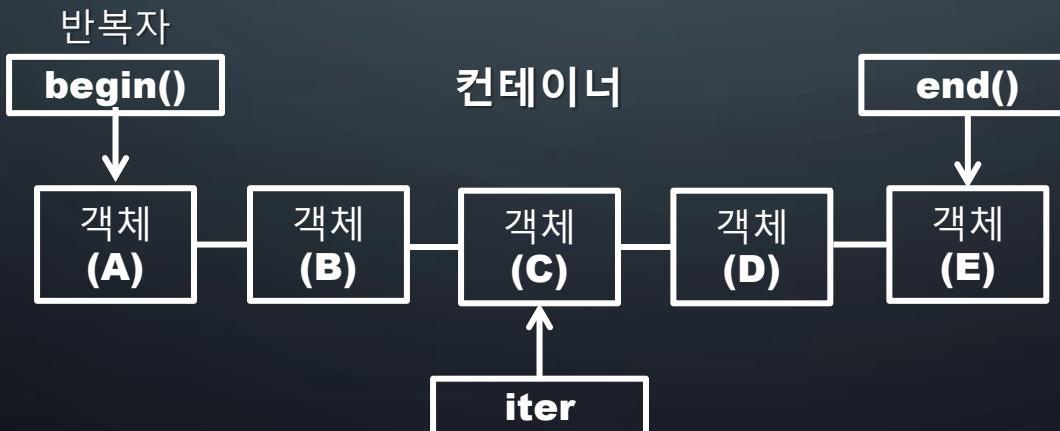
- 포인터와 비슷하다.
- 컨테이너의 저장된 원소를 순회하고 접근하는 일반화된 방법을 제공한다.
- 컨테이너와 알고리즘이 하나로 동작하게 묶어주는 인터페이스 역할을 한다.
- 컨테이너마다 자신만의 반복자를 가지고 있다.

## 특징

- 컨테이너 내부의 원소(객체)를 가리키고 접근할 수 있어야 한다.(\*연산자 제공)
- 다음 원소로 이동하고 컨테이너의 모든 원소를 순회할 수 있어야 한다.(++, !=, == 연산자 제공)

## 순차열

- 원소의 순서가 있는 집합을 의미한다.
- 순차열의 시작과 끝에서 반복자를 반환한다.
- 끝은 실제 원소가 아닌 끝을 표시한다. - 반개구간 **[begin, end)**



# 1. STL의 이해

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

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);
```

**vector<int>::iterator iter;** // 반복자 생성(아직 원소를 가리키지 않음)

```
for(iter = v.begin(); iter != v.end(); ++iter)
    cout << *iter << endl; // 반복자가 가리키는 원소를 역참조
```

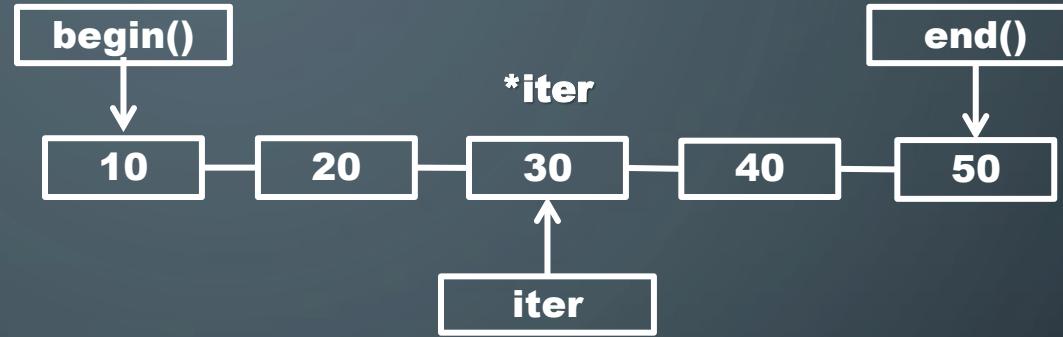
}

**v.begin()** : 컨테이너의 시작 원소를 가리키는 반복자를 반환

**v.end()** : 컨테이너의 끝 표시 반복자를 반환

**++iter** : 반복자를 다음 원소를 가리키도록 이동

**\*iter** : **iter**가 가리키는 원소(객체)를 반환(역참조)



# 1. STL의 이해

## 반복자의 범주

### 입력 반복자(input iterator)

- 현 위치의 원소를 한 번만 읽을 수 있는 반복자

### 출력 반복자(output iterator)

- 현 위치의 원소를 한번 만 쓸 수 있는 반복자.

### 순방향 반복자(forward iterator)

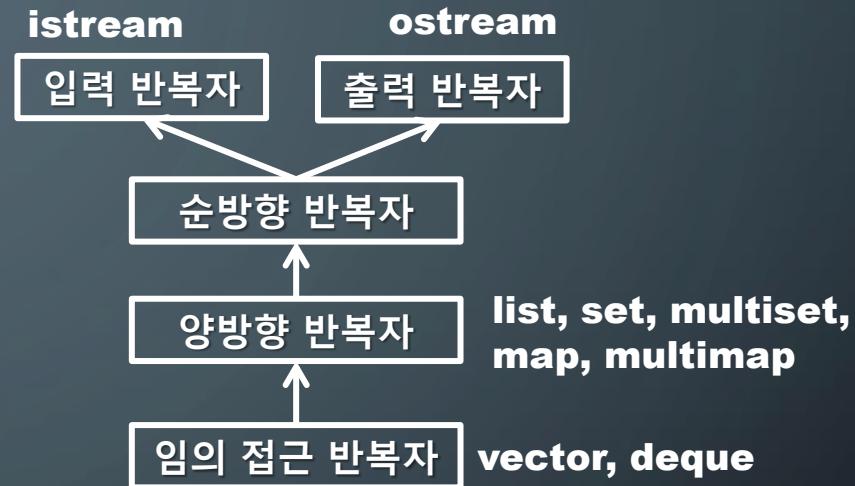
- 입력, 출력 반복자 기능에 순방향으로 이동(++)이 가능한 반복자

### 양방향 반복자(bidirectional iterator)

- 순방향 반복자 기능에 역방향으로 이동(--)이 가능한 반복자

### 임의 접근 반복자(random access iterator)

- 양방향 반복자 기능에 +, -, +=, -=, []연산이 가능한 반복자



# 1. STL의 이해

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

int main()
{
    deque<int> dq;

    dq.push_back(10);
    dq.push_back(20);
    dq.push_back(30);
    dq.push_back(40);
    dq.push_back(50);

    deque<int>::iterator iter = dq.begin(); // 시작 원소를 가리키는 반복자
    cout << iter[0] << endl; // [] 연산자
    cout << iter[1] << endl;
    cout << iter[2] << endl;
    cout << iter[3] << endl;
    cout << iter[4] << endl;
    cout << endl;

    iter += 2; // += 연산
    cout << *iter << endl;
    cout << endl;

    deque<int>::iterator iter2 = iter + 2; // + 연산을 통해서 임의의 접근을 수행했다.
    cout << *iter2 << endl;

    return 0;
}
```

**deque<int>::iterator iter2 = iter + 2;** // + 연산을 통해서 임의의 접근을 수행했다.

# 1. STL의 이해

## 알고리즘

- 순차열의 원소를 조사, 변경, 관리, 처리할 목적으로 구성해서 제공하고 있다.
- 한 쌍의 반복자([**begin**, **end**])이 필요하며 **순방향 반복자를 요구**한다.
- 몇몇의 알고리즘은 순방향이 아닌 **임의의 접근 반복자를 요구**하기도 한다.
- 같은 기능을 수행하는 알고리즘들이 오버로딩되어 많은 수를 이루고 있지만 그 범주는 **7**가지 정도 된다.
- 일반적이며 특정 컨테이너나 원소 타입에 종속적이지 않다.

## 알고리즘의 분류

- 원소를 수정하지 않는 알고리즘(**nonmodifying algorithms**)
- 원소를 수정하는 알고리즘(**modifying algorithms**)
- 제거 알고리즘(**removing algorithms**)
- 변경 알고리즘(**mutating algorithms**)
- 정렬 알고리즘(**sorting algorithms**)
- 정렬된 범위 알고리즘(**sorted range algorithms**)
- 수치 알고리즘(**numeric algorithms**)

# 1. STL의 이해

## Find 알고리즘 예제

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

int main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector(int)::iterator iter;
    iter = find(v.begin(), v.end(), 20); // [begin, end)에서 20찾기
    cout << *iter << endl;

    iter = find(v.begin(), v.end(), 100) // [begin, end)에서 100찾기
    if(iter == v.end()) // 100이 없으면 iter == v.end()가 된다.
        cout << "100이 없음" << endl;

    return 0;
}
```

# 1. STL의 이해

## 함수객체

- 클라이언트가 정의한 동작을 다른 구성 요소에 반영하려 할 때 사용한다.

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

int main()
{
    vector<int> v;
    v.push_back(50);
    v.push_back(30);
    v.push_back(10);
    v.push_back(40);
    v.push_back(20);

    sort(v.begin(), v.end(), less<int>()); // 오름차순 정렬
    for(vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    sort(v.begin(), v.end(), greater<int>()); // 내림차순 정렬
    for(vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

# 1. STL의 이해

## 어댑터

- 구성 요소의 인터페이스를 바꾼다.
- 컨테이너 어댑터, 반복자 어댑터, 함수 어댑터가 있다.

### 컨테이너 어댑터(**container adaptor**)

- **stack, queue, priority\_queue**

### 반복자 어댑터(**iterator adaptor**)

- **reverse\_iterator, back\_insert\_iterator, front\_insert\_iterator, insert\_iterator**

### 함수 어댑터(**function adapter**)

- **바인더(binder), 부정자(negator), 함수 포인터 어댑터(adator for pointers to functions)**

# 1. STL의 이해

## Stack의 예제

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

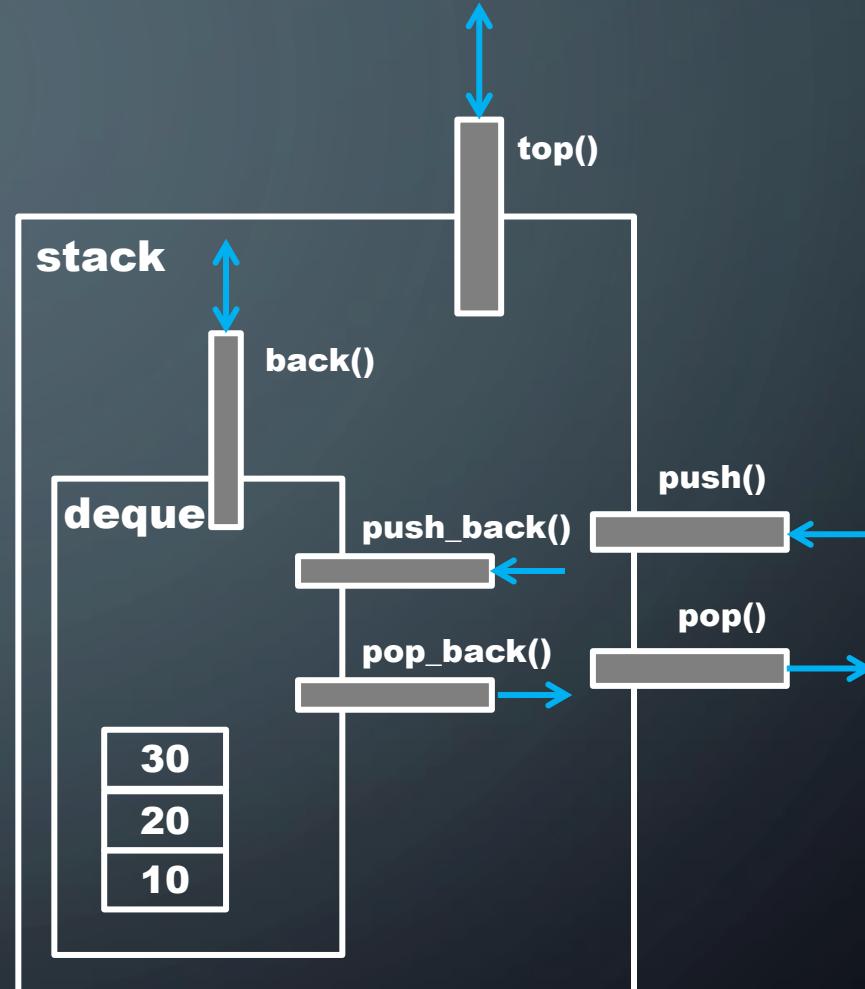
Int main()
{
    stack<int> st;

    st.push(10);
    st.push(20);
    st.push(30);

    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();

    if(st.empty())
        cout << "stack에 데이터 없음" << endl;

    return 0;
}
```



stack은 deque를 이용해서  
stack 컨테이너를 생성한다.

# 1. STL의 이해

## vector 컨테이너를 적용한 스택

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

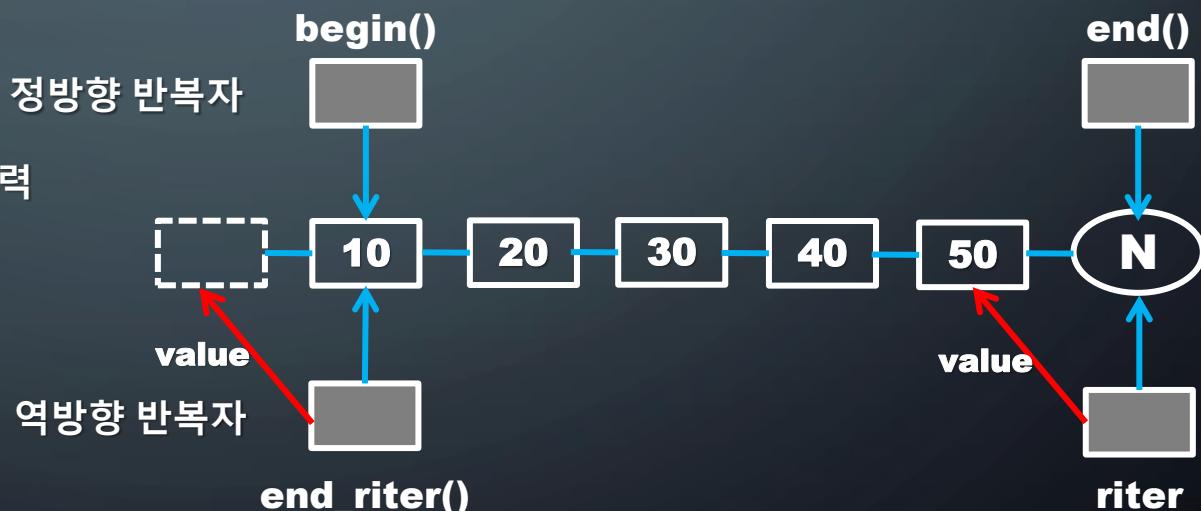
void main( )
{
    stack<int, vector<int> > st; // vector 컨테이너를 이용한 stack 컨테이너 생성

    st.push( 10 ); // 데이터 추가(입력)
    st.push( 20 );
    st.push( 30 );

    cout << st.top() << endl; // top 데이터 출력
    st.pop(); // top 데이터 삭제
    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();

    if( st.empty() ) // 스택이 비었는지 확인
        cout << "stack이 데이터 없음" << endl;
}
```

반복자는 가리키는 위치의 다음 위치의 값을 참조하기 때문에  
반복자의 위치와 참조하는 값은 다르다.



# 1. STL의 이해

## 역방향 반복자로 전환

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

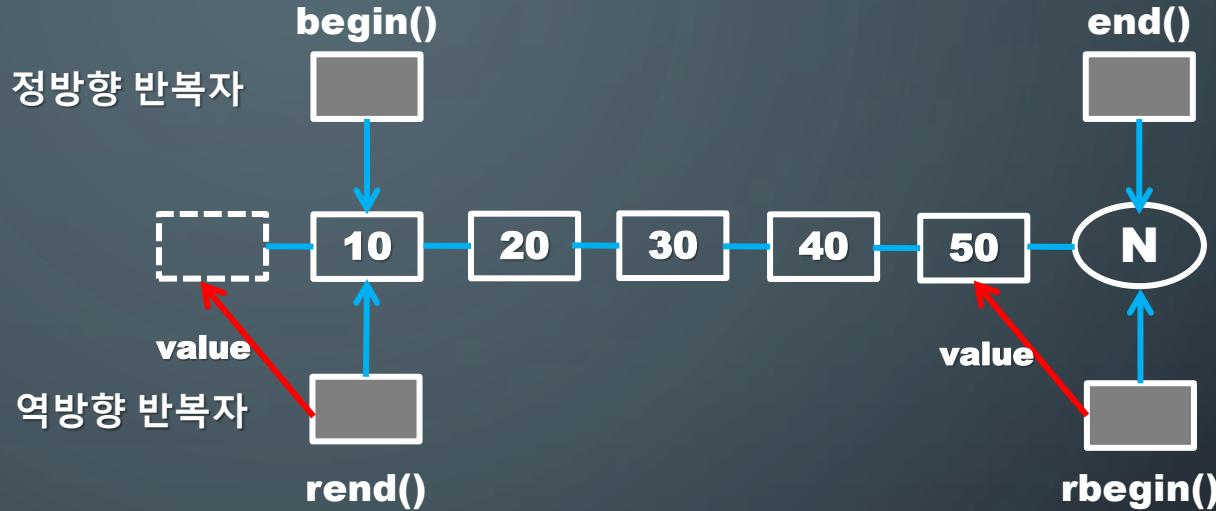
void main( )
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for( vector<int>::iterator iter= v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    //일반 반복자 iterator를 역방향 반복자 reverse_iterator로 변환
    reverse_iterator< vector<int>::iterator > riter(v.end());
    reverse_iterator< vector<int>::iterator > end_riter(v.begin());

    for( ; riter != end_riter ; ++riter )
        cout << *riter << " ";
    cout << endl;
}
```



# 1. STL의 이해

## vector의 역방향 반복자.

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

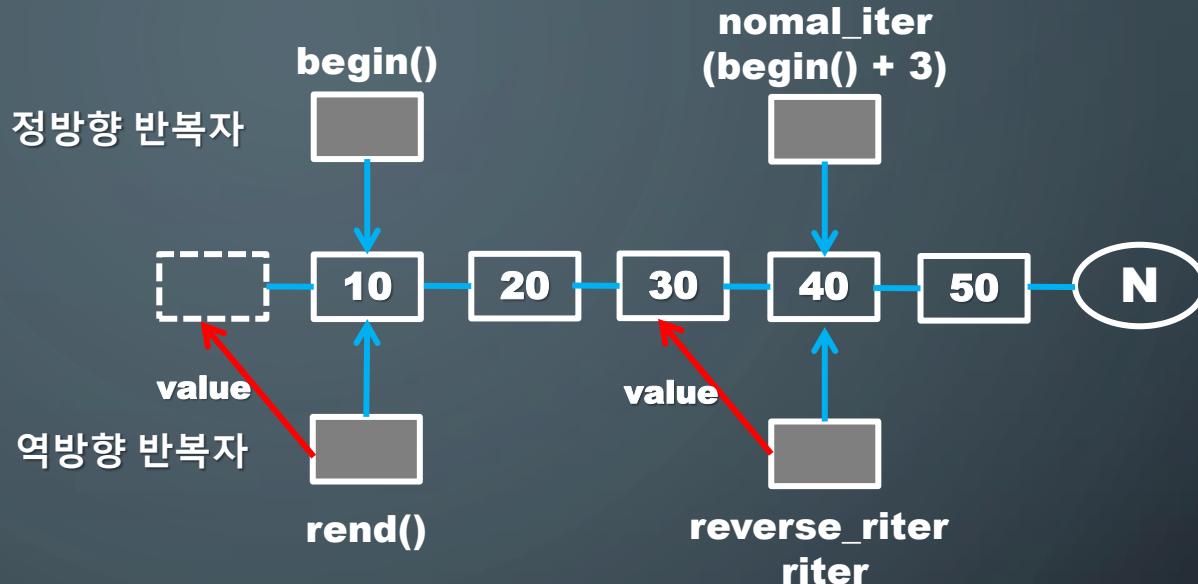
```
void main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);
```

```
for( vector<int>::iterator iter= v.begin(); iter != v.end(); ++iter)
    cout << *iter << " ";
cout << endl;
```

// STL 모든 컨테이너는 반복자 어댑터 **reverse\_iterator**를 **typedef** 타입으로 정의하며  
// **rbegin()**, **rend()**로 컨테이너의 역방향 반복자를 반환함.  
**vector<int>::reverse\_iterator riter(v.rbegin());**

```
for( ; riter != v.rend() ; ++riter )
    cout << *riter << " ";
cout << endl;
```



# **VECTOR** – 시퀀스 컨테이너

# 1. VECTOR – 시퀀스 컨테이너

## 특징

- 임의의 접근 반복자를 지원하는 배열 기반 컨테이너
- 원소가 하나의 메모리 블록에 연속하게 저장되어 있다.
- 배열의 특징인 **각 원소로의 접근이 빠르다는 장점을** 그대로 가지고 있다.
- 원소가 추가되거나 삽입될 때 메모리 재할당이 발생할 수도 있고 상당한 비용을 쓰게 된다.**
- 메모리 할당크기를 알 수 있는 `capacity()`와 예약할당으로 미리 크게 할당을 할 수 있는 `reserve()`함수를 제공한다.
- 삽입, 삭제, 반환 같은 것이 빈번하게 발생하는 프로그램이라면 사용하지 않는 것이 좋다.**

## 인터페이스

템플릿 형식

**Template<typename T, typename Allocator = allocator<T>>**

**class vector**

할당기

### 생성자

**vector v**

v는 빈 컨테이너이다.

**vector v(n)**

v는 기본값으로 초기화된 n개의 원소를 갖는다.

**vector v(n, x)**

v는 x값으로 초기화된 n개의 원소를 갖는다.

**vector v(v2)**

v는 v2 컨테이너의 복사본이다.(복사 생성자 호출).

**vector v(b, e)**

v는 반복자 구간 [b, e)로 초기화 된 원소를 갖는다.

# 1. VECTOR – 시퀀스 컨테이너

## 연산자

|                    |  |
|--------------------|--|
| <b>v1 == v2</b>    | v1과 v2의 모든 원소가 같은가?(bool 형식)                                       |
| <b>v1 != v2</b>    | v1과 v2의 모든 원소 중 하나라도 다른 원소가 있는가?(bool 형식)                          |
| <b>v1 &lt; v2</b>  | 문자열 비교처럼 v2가 v1보다 큰가?(bool 형식)                                     |
| <b>v1 &lt;= v2</b> | 문자열 비교처럼 v2가 v1보다 크거나 같은가?(bool 형식)                                |
| <b>v1 &gt; v2</b>  | 문자열 비교처럼 v1이 v2보다 큰가?(bool 형식)                                     |
| <b>v1 &gt;= v2</b> | 문자열 비교처럼 v1이 v2보다 크거나 같은가?(bool 형식)                                |
| <b>v[i]</b>        | v의 i번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있으며 범위 점검이 없다.) |

## 멤버 형식

|                               |                                 |
|-------------------------------|---------------------------------|
| <b>allocator_type</b>         | 메모리 관리자 형식                      |
| <b>const_iterator</b>         | <b>const</b> 반복자 형식             |
| <b>const_pointer</b>          | <b>Const value_type*</b> 형식     |
| <b>const_reference</b>        | <b>Const value_type&amp;</b> 형식 |
| <b>const_reverse_iterator</b> | <b>Const</b> 역 반복자 형식           |
| <b>difference_type</b>        | 두 반복자 차이의 형식                    |
| <b>iterator</b>               | 반복자 형식                          |

# 1. VECTOR – 시퀀스 컨테이너

| 멤버 함수                    |   |
|--------------------------|---|
| <b>v.assign(n, x)</b>    | <b>v</b> 에 <b>x</b> 값으로 <b>n</b> 개의 원소를 할당한다.                                     |
| <b>v.assign(b, e)</b>    | <b>v</b> 를 반복자 구간 <b>[b, e)</b> 로 할당한다.   |
| <b>v.at(i)</b>           | <b>v</b> 의 <b>i</b> 번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있으며 범위 점검을 포함) |
| <b>v.back()</b>          | <b>v</b> 의 마지막 원소를 참조한다. ( <b>const</b> , 비 <b>const</b> 버전이 있음)                  |
| <b>p=v.begin()</b>       | <b>p</b> 는 <b>v</b> 의 첫 원소를 가리키는 반복자( <b>const</b> , 비 <b>const</b> 버전이 있음)       |
| <b>x=v.capacity()</b>    | <b>x</b> 는 <b>v</b> 에 할당된 공간의 크기  |
| <b>v.clear()</b>         | <b>v</b> 의 모든 원소를 제거한다.   |
| <b>v.empty()</b>         | <b>v</b> 가 비었는지 조사한다.   |
| <b>p=v.end()</b>         | <b>p</b> 는 <b>v</b> 의 끝을 표시하는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음)         |
| <b>q=v.erase(p)</b>      | <b>p</b> 가 가리키는 원소를 제거한다. <b>q</b> 는 다음 원소를 가리킨다.                                 |
| <b>q=v.erase(b,e)</b>    | 반복자 구간 <b>[b, e)</b> 의 모든 원소를 제거한다. <b>q</b> 는 다음 원소를 가리킨다.                       |
| <b>v.front()</b>         | <b>v</b> 의 첫 번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있음)                   |
| <b>q=v.insert(p, x)</b>  | <b>p</b> 가 가리키는 위치에 <b>x</b> 값을 삽입한다. <b>q</b> 는 삽입한 원소를 가리키는 반복자.                |
| <b>v.insert(p, n, x)</b> | <b>p</b> 가 가리키는 위치에 <b>n</b> 개의 <b>c</b> 값을 삽입한다.                                 |

# 1. VECTOR – 시퀀스 컨테이너

| 멤버 함수                    |   |
|--------------------------|---|
| <b>v.insert(p, b, e)</b> | <b>p</b> 가 가리키는 위치에 반복자 구간 <b>[b, e)</b> 의 원소를 삽입한다.                                |
| <b>x=v.max_size()</b>    | <b>x</b> 는 <b>v</b> 가 담을 수 있는 최대 원소의 개수다(메모리의 크기)                                   |
| <b>v.pop_back()</b>      | <b>v</b> 의 마지막 원소를 제거한다.  |
| <b>v.push_back(x)</b>    | <b>v</b> 의 끝에 <b>x</b> 를 추가한다.  |
| <b>p=v.regin()</b>       | <b>p</b> 는 <b>v</b> 의 역 순차열의 첫 원소를 가리키는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음) |
| <b>p=v.rend()</b>        | <b>p</b> 는 <b>v</b> 의 역 순차열의 끝을 표시하는 반복자( <b>const</b> , 비 <b>const</b> 버전이 있음)     |
| <b>v.reserve(n)</b>      | <b>n</b> 개의 원소를 저장할 공간을 예약한다.   |
| <b>v.resize(n)</b>       | <b>v</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 기본값으로 초기화 한다.                           |
| <b>v.resize(n, x)</b>    | <b>v</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 <b>x</b> 값으로 초기화 한다.                    |
| <b>v.size()</b>          | <b>v</b> 의 원소의 개수다.   |
| <b>v.swap(v2)</b>        | <b>v</b> 와 <b>v2</b> 를 <b>swap</b> 한다.  |

# 1. VECTOR – 시퀀스 컨테이너

## vector의 주요 구성도

**insert():** 삽입



**push\_back, size(), []연산자 사용 예제..**

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

void main( )
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(int i = 0 ; i < v.size() ; ++i)
        cout << v[i] << endl;
}
```

**vector<int> :** int 형식의 원소를 저장하는 컨테이너 **v**를 생성  
**v.push\_back(x) :** **v**에 **x**를 추가한다.  
**v.size() :** **v**의 원소의 개수를 반환한다.  
**v[i] :** **v**의 **i**번째 원소를 참조한다.

# 1. VECTOR – 시퀀스 컨테이너

**size()** 함수의 경우 **unsigned int** 타입을 반환 **size\_type**을 쓰는 것을 권장한다.

## Size\_type 사용 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << endl;

    cout << typeid(vector<int>::size_type).name() << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

## vector 크기 반환 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;

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

**size()**, **max\_size()**와 달리 **capacity()**는 배열기반인 **vector**에만 존재한다. 데이터가 추가 될 때마다 배열은 크기가 증가해야 하기 때문에 재 할당과 원소 복사를 반복한다. 이런 과정을 조금이라도 줄이기 위해 한번 할당 할 때마다 조금 여유를 가지면서 할당을 한다. 이 때, **capacity**는 할당된 값을 가지고 있는 것이다.

# 1. VECTOR – 시퀀스 컨테이너

## Capacity() 이해를 돋는 예제

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

    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(10);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(20);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(30);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(40);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(50);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(60);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(70);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(80);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(90);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

```
size: 0 capacity: 0
size: 1 capacity: 1
size: 2 capacity: 2
size: 3 capacity: 3
size: 4 capacity: 4
size: 5 capacity: 6
size: 6 capacity: 6
size: 7 capacity: 9
size: 8 capacity: 9
size: 9 capacity: 9
10 20 30 40 50 60 70 80 90
```

# 1. VECTOR – 시퀀스 컨테이너

## capacity()의 동작

size:0  
capacity:0

size:1  
capacity:1

10

size:2  
capacity:2

10  
20

size:3  
capacity:3

10  
20  
30

size:4  
capacity:4

10  
20  
30  
40

최초  
메모리 할당

메모리  
재할당과  
원소복사

메모리  
재할당과  
원소복사

메모리  
재할당과  
원소복사

size:5  
capacity:6

10  
20  
30  
40  
50

size:6  
capacity:6

10  
20  
30  
40  
50  
60

size:7  
capacity:9

10  
20  
30  
40  
50  
60  
70

size:8  
capacity:9

10  
20  
30  
40  
50  
60  
70  
80

size:9  
capacity:9

10  
20  
30  
40  
50  
60  
70  
80  
90

메모리  
재할당과  
원소복사

메모리  
재할당과  
원소복사

# 1. VECTOR – 시퀀스 컨테이너

**reserve()**을 사용 메모리를 예약하는 예제..

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

    v.reserve(8);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(10);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(20);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(30);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(40);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(50);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(60);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(70);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(80);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;
    v.push_back(90);
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

**reserve()**를 통해 메모리를 미리 할당 받았기 때문에 그 범위가 넘어가지 않는 이상 **capacity**는 할당된 크기의 값을 가지고 있다.

# 1. VECTOR – 시퀀스 컨테이너

생성자를 통한 **size** 확보 예제..

```
void main( )
{
    vector<int> v1(5); //0으로 초기화된 size가 5인 컨테이너
    v1.push_back(10); //10 추가
    v1.push_back(20); //20 추가
    v1.push_back(30); //30 추가
    v1.push_back(40); //40 추가
    v1.push_back(50); //50 추가

    for(vector<int>::size_type i = 0 ; i < v1.size() ; ++i)
        cout << v1[i] << " ";

    cout << endl;

    vector<int> v2(5); //0으로 초기화된 size가 5인 컨테이너
    v2[0] = 10; // 0에서 10로 수정
    v2[1] = 20; // 0에서 20로 수정
    v2[2] = 30; // 0에서 30로 수정
    v2[3] = 40; // 0에서 40로 수정
    v2[4] = 50; // 0에서 50로 수정

    for(vector<int>::size_type i = 0 ; i < v2.size() ; ++i)
        cout << v2[i] << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

생성자에 원소의 크기와 최소값을 지정하는 예제

```
void main()
{
    vector<int> v1(5); //기본값 0으로 초기화된 size가 5인 컨테이너

    for (vector<int>::size_type i = 0; i < v1.size(); ++i)
        cout << v1[i] << " ";

    cout << endl;

    vector<int> v2(5, 0); //지정값 0으로 초기화된 size가 5인 컨테이너

    for (vector<int>::size_type i = 0; i < v2.size(); ++i)
        cout << v2[i] << " ";

    cout << endl;

    vector<int> v3(5, 10); //지정값 10으로 초기화된 size가 5인 컨테이너

    for (vector<int>::size_type i = 0; i < v3.size(); ++i)
        cout << v3[i] << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

**resize()**이용해서 컨테이너 크기 변경하는 예제..

```
void main()
{
    vector<int> v(5); //기본값 0으로 초기화된 size가 5인 컨테이너

    v[0] = 10;
    v[1] = 20;
    v[2] = 30;
    v[3] = 40;
    v[4] = 50;

    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

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

    v.resize(10); //기본값 0으로 초기화된 size가 10인 컨테이너로 확장
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

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

    v.resize(5); //size가 5인 컨테이너로 축소 capacity는 변화 없음.
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

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

# 1. VECTOR – 시퀀스 컨테이너

**clear()와 empty()를 사용하는 예제..**

```
void main( )
{
    vector<int> v(5);

    v[0] = 10;
    v[1] = 20;
    v[2] = 30;
    v[3] = 40;
    v[4] = 50;

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

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

    v.clear(); // v를 비운다.
    cout <<"size: "<< v.size() <<" capacity: " << v.capacity() <<endl;

    if( v.empty() ) // v가 비었는가?
    {
        cout << "v에 원소가 없습니다." << endl;
    }
}
```

# 1. VECTOR – 시퀀스 컨테이너

Swap()을 이용해서 capacity를 0으로 만드는 예제..

```
void main( )
{
    vector<int> v(5);

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

    vector<int>().swap(v); // 기본 생성자로 만든 vector컨테이너와 v 컨테이너를 swap한다.

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

# 1. VECTOR – 시퀀스 컨테이너

**swap()**을 이용한 값 교체 예제..

```
void main( )
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);

    vector<int> v2;
    v2.push_back(100);
    v2.push_back(200);
    v2.push_back(300);

    for(vector<int>::size_type i = 0 ; i < v1.size() ; ++i)
        cout << v1[i] << ", " << v2[i] << endl;

    cout << endl;

    v1.swap(v2); // v1과 v2를 swap합니다.
    for(vector<int>::size_type i = 0 ; i < v1.size() ; ++i)
        cout << v1[i] << ", " << v2[i] << endl;

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

**front()와 back()**을 이용한 첫 번째 원소와 마지막 원소를 출력한 예제..

```
void main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

    cout << endl;

    cout << v[0] << ", " << v.front() << endl; // 첫 번째 원소
    cout << v[4] << ", " << v.back() << endl; // 마지막 원소
}
```

# 1. VECTOR – 시퀀스 컨테이너

**front()**와 **back()** 참조를 이용해 원소를 수정하는 예제..

```
void main( )
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;

    v.front() = 100; // 첫 번째 원소를 100으로
    v.back() = 500; // 마지막 원소를 500으로

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

## 연산자와 at() 멤버 함수 사용 예제..

```
void main( )
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;

    v[0] = 100; //범위 점검 없는 0 index 원소의 참조
    v[4] = 500; //범위 점검 없는 4 index 원소의 참조

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;

    v.at(0) = 1000; //범위 점검 있는 0 index 원소의 참조
    v.at(4) = 5000; //범위 점검 있는 4 index 원소의 참조

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v.at(i) << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

**at()** 멤버 함수의 **out\_of\_range** 범위 오류 예제..

```
void main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    try
    {
        cout << v.at(0) << endl;
        cout << v.at(3) << endl;
        cout << v.at(6) << endl; //throw out_of_range 예외
    }
    catch (out_of_range &e)
    {
        cout << e.what() << endl;
    }
}
```

# 1. VECTOR – 시퀀스 컨테이너

**assign()** 멤버 함수를 사용한 원소 값 할당 예제..

```
void main()
{
    vector<int> v(5, 1); // 초기값 1의 5개의 원소를 갖는 컨테이너 생성

    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

    cout << endl;

    v.assign(5, 2); // 5개의 원소값을 2로 할당
    cout << v.size() << ',' << v.capacity() << endl;

    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

반복자를 이용한 출력 예제..

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

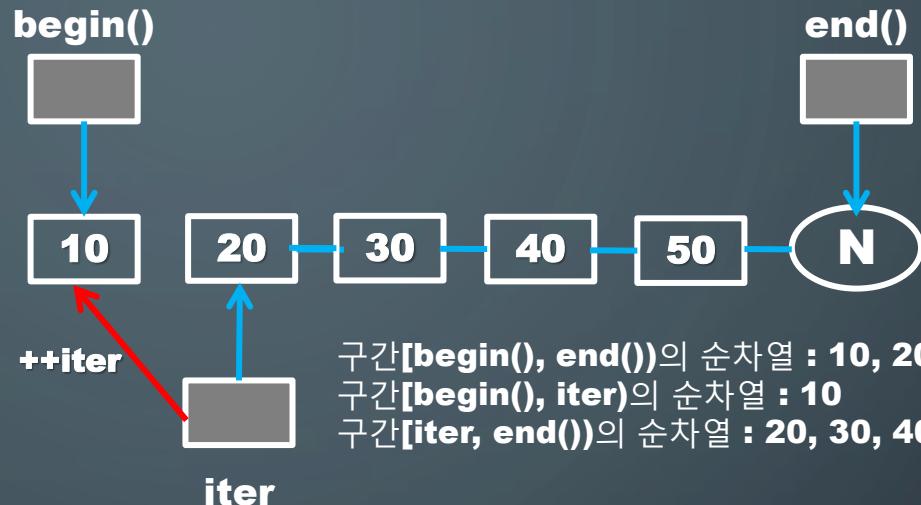
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(vector<int>::size_type i = 0 ; i < v.size() ; ++i)
        cout << v[i] << " ";

    cout << endl;

    for(vector<int>::iterator iter = v.begin(); iter != v.end() ; ++iter)
        cout << *iter << " ";

    cout << endl;
}
```



구간[begin(), end())의 순차열 : 10, 20, 30, 40, 50  
구간[begin(), iter)의 순차열 : 10  
구간[iter, end())의 순차열 : 20, 30, 40, 50

**vector<int>::iterator** : 반복자 클래스다. **Vector**내에 정의 되어 있다.  
**vector<int>::iterator iter** : iter라는 반복자 객체를 생성한다.  
**v.begin()** : v의 첫 번째 원소를 가리키는 반복자를 반환한다.  
**v.end()** : v의 끝을 표시하는 반복자를 반환  
**Iter != v.end()** : iter가 끝을 표시하는 반복자가 아니면 참.  
**++iter** : 반복자를 다음 원소를 가리키도록 이동한다.  
**\*iter** : 반복자가 가리키는 원소의 참조

# 1. VECTOR – 시퀀스 컨테이너

반복자에 **+ =**, **- =** 연산하는 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

vector<int>::iterator iter = v.begin(); // 시작 원소(10)를 가리킨다.
cout << *iter << endl;

iter += 2; // +2한 위치의 원소(30)를 가리킨다.
cout << *iter << endl;

iter -= 1; // -1한 위치의 원소(20)를 가리킨다.
cout << *iter << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

**vector<int>::iterator**와 **const**반복자 **vector<int>::const\_iterator**의 비교 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter = v.begin();
    vector<int>::const_iterator citer = v.begin();

    cout << *iter << endl; //가리키는 원소의 참조
    cout << *citer << endl; //가리키는 원소의 상수 참조

    cout << *++iter << endl; //다음 원소로 이동한 원소의 참조
    cout << *++citer << endl; //다음 원소로 이동한 원소의 상수 참조

    *iter = 100; // 일반 반복자는 가리키는 원소를 변경할 수 있음
    /*citer = 100; // 상수 반복자는 가리키는 원소를 변경할 수 없음
}
```

**vector<int>::const\_iterator**는 **const int\*** 처럼 동작하여 읽기가 가능하지만, **const vector<int>::iterator**은 **int\* const**처럼 동작하기 때문에 참조의 변형자체가 안되므로 반복자를 이동할 수 없게 된다.

# 1. VECTOR – 시퀀스 컨테이너

Iterator와 reverse\_iterator를 사용하는 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter; //정방향 반복자
    vector<int>::reverse_iterator riter; //역방향 반복자

    for( iter = v.begin(); iter != v.end() ; ++iter)
        cout << *iter << " ";

    cout << endl;

    for( riter = v.rbegin(); riter != v.rend() ; ++riter)
        cout << *riter << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

## Insert() 멤버 함수 사용 예제..

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

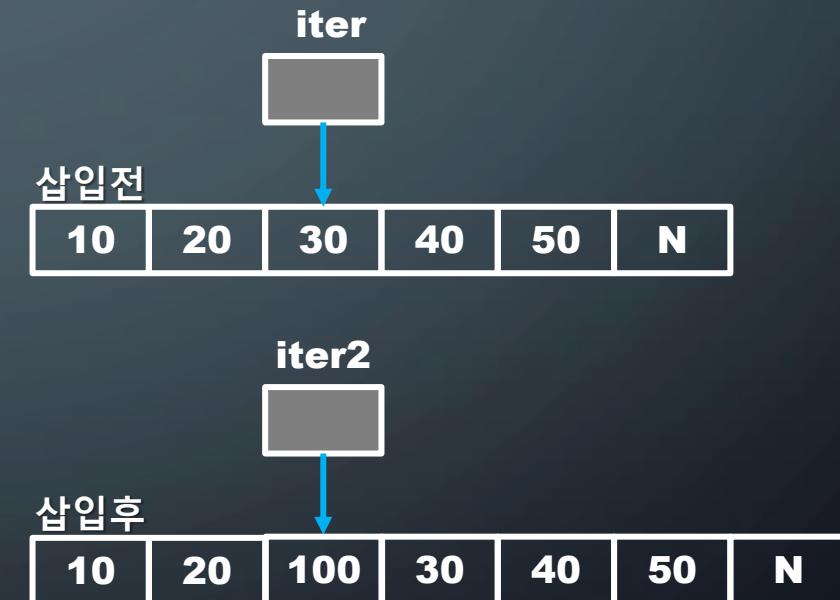
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter = v.begin() + 2;
    vector<int>::iterator iter2;

    // iter가 가리키는 위치에 정수 100을 삽입.
    // iter2는 삽입한 정수를 가리키는 반복자.
    iter2 = v.insert(iter, 100);

    for (iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";

    cout << endl;
    cout << "iter2: " << *iter2 << endl;
}
```



# 1. VECTOR – 시퀀스 컨테이너

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter = v.begin() + 2;

    // iter가 가리키는 위치에 정수 100을 3개 삽입.
    v.insert(iter, 3, 100);

    for (iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    /////////////////////////////////
    vector<int> v2;
    v2.push_back(100);
    v2.push_back(200);
    v2.push_back(300);

    iter = v2.begin() + 1;
    // iter가 가리키는 위치에 [v.begin(), v.end()) 구간의 원소를 삽입.
    v2.insert(iter, v.begin(), v.end());

    for (iter = v2.begin(); iter != v2.end(); ++iter)
        cout << *iter << " ";
    cout << endl;
}
```

**v.insert(iter, 3, 100) :** iter가 가리키는 위치의 v 컨테이너의 3개의 100 값을 삽입한다.  
**v2.insert(iter, v.begin(), v.end()) :** iter가 가리키는 위치의 v2 컨테이너에 구간 [v.begin(), v.end())의 원소를 삽입한다.

# 1. VECTOR – 시퀀스 컨테이너

## erase() 멤버 함수 사용 예제..

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

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter;
    vector<int>::iterator iter2;

    for (iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";

    cout << endl;

    iter = v.begin() + 2;
    // iter가 가리키는 위치의 원소를 제거합니다. iter2는 다음 원소 40
    iter2 = v.erase(iter);

    for (iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";

    cout << endl;

    // [v.begin()+1, v.end()) 구간의 원소를 제거합니다.
    iter2 = v.erase(v.begin() + 1, v.end()); // iter2는 다음 원소 v.end()

    for (iter = v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

반복자로 동작하는 생성자와 **assign()** 멤버 함수

```
void main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int> v2(v.begin(), v.end()); // 순차열 [v.begin(), v.end())로 v2를 초기화.
    vector<int>::iterator iter;

    for (iter = v2.begin(); iter != v2.end(); ++iter)
        cout << *iter << " "; // v2 출력

    cout << endl;

    vector<int> v3;
    v3.assign(v.begin(), v.end()); // v3에 순차열 [v.begin(), v.end())을 할당.

    for (iter = v3.begin(); iter != v3.end(); ++iter)
        cout << *iter << " "; // v3 출력

    cout << endl;
}
```

# 1. VECTOR – 시퀀스 컨테이너

## vector와 vector의 비교

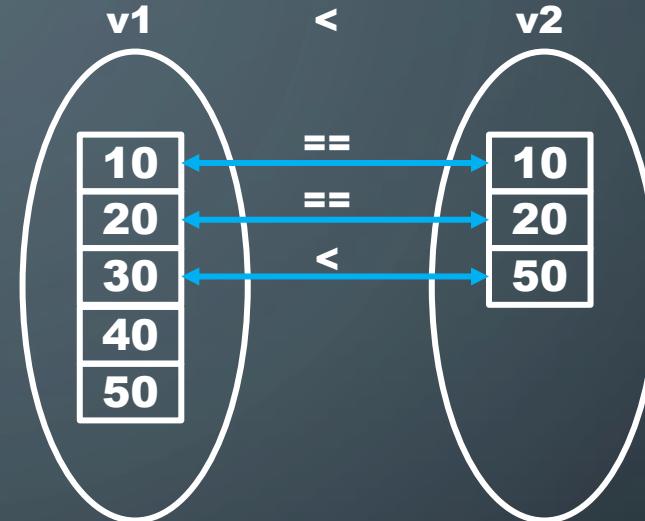
```
void main()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    vector<int> v2;
    v2.push_back(10);
    v2.push_back(20);
    v2.push_back(50);

    if (v1 == v2)
        cout << "v1 == v2" << endl;

    if (v1 != v2)
        cout << "v1 != v2" << endl;

    if (v1 < v2)
        cout << "v1 < v2" << endl;
}
```



v1 == v2 : v1과 v2의 모든 원소가 같다면 참 아니면 거짓이다.  
v1 != v2 : v1과 v2의 모든 원소가 같다면 거짓 아니면 참이다.  
v1 < v2 : v1과 v2의 순차열의 원소를 하나씩 순서대로 비교하여 v2의 원소가 크다면 참 아니면 거짓이다.

# 1. VECTOR – 시퀀스 컨테이너

## 학습과제

미로찾기 맵을 `vector`로 생성, 관리, 출력하는 프로그램을 만들어 보자.

→ 미로찾기 맵을 그리는 부분까지만 만들면 됨.