**▣ 6장 STL 컨테이너**

**6.2 Vector (벡터)**

- 컨테이너에 데이터가 삽입 될수록 메모리는 늘어나게 됩니다. 연속된 메모리 기반이므로 메모리가 커지면 기존 메모리를 삭제하고 새로운 메모리를 재할당해주어야 합니다. 그러므로 컨테이너요소의 개수가 유동적인 곳에서는 비효율적이다.

**능력**

-시퀀스 컨테이너, 램덤 액세스 지원, 연속 메모리 기반 컨테이너.

- size(), empty(), max\_size(): 사이즈 관련된 함수 제공.

- capacity() : 메모리 안에 vector가 메모리의 재할당 없이 가질 수 있는 원소의 개수를 반환. 초과한다면 내부메모리를 재할당해야 함.

- vector 용량이 중요한 이유 : 따라서 속도가 목적이라면 많은 용량을 확보하는 것이 유리.

- 재할당의 원소의 모든 레퍼런스와 포인터, 반복자를 무효화시킨다.

- 재할당은 시간을 필요로 함.

- 재할당 피하는 방법

1. 생성자를 이용하여 메모리공간(capacity)와 크기(size)를 미리 확보, 초기화 합니다.

- std::vector<T> **v**(5);

2. reserve()을 이용하여 메모리를 미리 확보합니다.

- std::vector <int> v; v.**reserve**(50);

**동작**

- 생성, 복사, 소멸 동작을 할 수 있다.

- 수정하지 않는 동작(size, empty, max\_size, capacity, reserve, 비교연산자)

- 할당 관련 동작(=, assign, swap)

**- 원소 액세스**

- 원소를 직접적으로 액세스하기 위한 동작

예) c.at(idx), c[idx], c.front(), c.back()

- 반복자 함수들: begin(), end, rbegin(), rend()

- 원소의 삽입 및 제거: insert(), push\_back(), pop\_back(), erase(), resize(), clear()

* 1. vector<bool>클래스

- 원소에 대해서 1비트만 사용하는 특수한 버전

- 특별한 비트 연산 제공(비트나 플래스(flag)를 간편하게 다룰 수 있음)

- bool 타입을 위한 vector<>의 특화된 버전

c.flip() : 모든 boolean 원소들을 반전한다(모든 비트의 보수)

m[idx].flip() : 인덱스가 idx인 원소를 반전시킨다(한 비트에 대한 보수)

m[idx] = val : val 을 인덱스가 idx인 boolean 원소에 할당한다.

m[idx1] = m[idx2] : 인덱스가 idx2인 원소의 값을 인덱스가 idx인 boolean 원소에 할당한다.

**6.3 Deque (덱)**

- 동적 배열에 원소들을 관리. 램덤 액세스 지원. 대부분 인터페이스가 vector와 유사.

- 양방향에 대해서 개방향. 시퀀스 컨테이너, 연속 메모리 기반 컨테이너이므로 임의 접근 반복자 제공

- “분리된 메모리 블록의 집합”방식으로 구현 (내부적으로 여러 개의 메모리 블럭을 사용함)

(요소의 추가, 삭제가 앞과 뒤에서 발생해도 vector나 string 처럼 메모리를 재할당하고 컨테이너의 모든 요소를 복사할 필요 없이 새로운 메모리 단위로 할당하여 새 요소만 추가하면 된다.)

**능력 (vector과 차이)**

- 원소들의 컨테이너의 앞과 끝에 삽입 및 삭제하는 과정이 빠르다.

- 하나이상의 간접성을 이용하기 때문에 원소의 액세스 시간과 반복자의 이동시간이 느리다.(분리된 메모리 블록을 이용하기 때문)

- 반복자는 반드시 스마트 포인터여야 한다. 이유는, 다른 메모리 블록으로 이동할 수 있어야 하기 때문

- 하나 이상의 메모리 블록을 사용하기 때문 vector에 비해 더 많은 원소를 가질 수 있다.

- 원소의 삽입, 삭제가 이루어 지는 순간, 원소를 참조하고 있던 모든 포인터와 레퍼런스, 반복자들을 무효화시킨다. (맨앞, 맨뒷부분에 삽입 및 제거의 경우는 예외)

**vector와 동일적용**

- 컨테이너 중간에 원소를 삽입 및 제거하는 동작이 느림.

- 반복자는 램덤 액세스를 지원.

**deque를 선호하는 경우**

- 컨테이너 앞쪽, 끝쪽으로 삽입, 삭제가 이루어지는 경우

- 컨테이너 원소들을 참조하지 않는 경우

- 컨테이너가 더 이상 사용되지 않을 경우 메모리가 해제되기를 원하는 경우

**vector와 동작이 다른 경우**

- 용량에 관한 함수(capacity(), reserve())를 제공하지 않음.

- 앞/끝부분에 직접적으로 원소를 추가, 삭제할 수 있는 함수제공: push\_back(), push\_front()

**6.4 List (리스트)**

- 시퀀스 컨테이너이며 노드기반의 컨테이너이다. 데이터의 삽입삭제가 시퀀스 중간에 자주 발생할 때 사용하면 좋은 컨테이너이다.

- 데이터를 앞뒤에 추가할 수 있는 push\_front(), push\_back() 함수제공.

**동작**

- 원소 액세스: 램덤 액세스를 지원하지 않으므로 원소를 직접적으로 액세스하기 위해 front(), back()을 제공.

- 양방향 반복자 제공(원소를 역으로 순회 할 수 있는 기능이 추가된 전방향 반복자로 모든 기능동일)

cf) list는 '연속 메모리 기반 컨테이너'가 아니므로 operator[]연산자는 가지고 있지 않고 반복자를 통해서만 컨테이너의 요소들에 접근할 수 있습니다. list는 양방향 반복자를 제공합니다. 그래서 반복자에 산술 연산은 불가능하다.(산술 연산이 가능한 반복자는 '임의 접근 반복자'뿐입니다. '연속 메모리 기반 컨테이너'가 임의 접근 반복자를 제공합니다.) 8개의 STL 컨테이너 모두 양방향 반복자 이상을 제공하며 이 중 '연속 메모리 기반 컨테이너'인 string, vector, deque는 산술 연산이 가능한 임의 접근 연산자를 제공합니다.

- 원소의 삽입 및 제거

: deque가 제공하는 모든 함수 제공, 추가적으로 remove(), remove\_if()가 제공.

(원소 제거시 remove 멤버함수를 사용. 포인터만 조작하기 때문에 remove알고리즘에 비해 빠르게 동작)

- 특별하게 수정하는 동작 : unique(), splice(), sort(), merge(), reverse()

**vector, deque와 비교시 차이점**

- 랜덤액세스를 지원하지 않아 속도 느림.

- 어떤 위치건 삽입, 삭제가 빠르다. (내부적으로 포인터 값들만 조작하면 되기 때문)

- 삽입과 삭제 동작이 모든 포인터와 레퍼런스, 반복자를 무효화 시키지 않는다.

**스플라이스(Splice)함수**

- 원소 또는 원소의 범위에 대해서 순서를 변경해 주거나 새롭게 링크를 걸어주는 멤버 함수.

- splice(pos, c2): c2의 모든 원소들을 c1의 pos위치 앞으로 이동한다.

예) c1.splice(pos, c2, c2pos); c2의 c2pos에 있는 원소를 c1의 pos위치 앞으로 이동한다.

**예외핸들링**

- STL컨테이너중 강력한 예외 안정성을 제공, 대부분의 성공 또는 아무런 영향을 받지 않는다. 보장하지 않는 것은 할당연산자와 sort() 멤버함수 뿐이다. (기본적인 안정성만 제공함: 리소스릭 방지)

**6.5 Set, MultiSet**

- Set의 특징은 유니크(unique)한 데이터(key) 셋의 균형 2진 트리로 저장합니다. Multiset은 같은 데이터(key) 값을 가질 수 있는 set입니다. 그래서 set과 multiset은 데이터(=key) 값을 빠르게 검색하고자 할 때 사용하면 좋은 컨테이너입니다.

- 자신들의 원소를 주어진 정렬 기준에 따라 자동적으로 정렬하는 컨테이너.

**정렬기준은 strict weak ordering에 부합되어야 한다.**

**1. 반드시 반대칭적(antisymmetric)여야만 한다.**

- <연산자에 대해서 만약 x < y가 true라면 y < x는 false만 가져야 한다.

조건자 op() 대해서 만약 op(x, y)가 tue라면 op(y, x)가 false만 가져야 한다는 것을 의미.

**2. 반드시 추이적(transitive)이여만 한다.**

- <연산자에 대해서 만약 x<y가 true이고 y<z이 true라면 x<z도 true여야만 한다. 조건자 op도 동일.

**3. 반드시 동등함에 대해서 추이적이여야만 한다.**

- 만약 a와 b가 같고, b와 c가 같을 경우, a와 c가 같다는 성립이 되어야 한다.

**4. 반드시 비반사적(irreflexive)이여야만 한다.**

- 만약 <연산자에 대해서, x < x 라면 항상 false를 보장

**능력**

- 균형 바이너리 트리(balanced binary tree)처럼 구현됨.

- 자동정렬의 장점은 특정값을 검색할 경우 바이너리 트리가 좋은 성능을 보장한다.

단점은 원소의 값을 직접적으로 수정할 수 없다. 이유, 값이 수정되면 순서가 깨지기 때문, 원소의 값을 변경하기 위해서 예전의 값을 제거하고, 새로운 값으로 변경된 원소를 삽입해야만 한다.

- 직접적인 원소 액세스를 허락하지 않는다.

- 반복자를 통하여 간접적으로 액세스하는 것도 제한을 받음. (반복자의 측면에서 보면 원소의 값은 상수)

**동작**

정렬기준의 2가지 방식

1. 템플릿의 파라미터로 넣는 방식이다.

예) std::set<int, std::**greater**<int> > coll;

2. 생성자의 파라미터로 넣는 방식이다.

- 정렬기준을 런타임시에 처리하고 싶은 경우, 또는 같은 데이터 타입에 대해서 다른 정렬 기준이 필요한 경우에 유용하다.

**수정하지 않는 동작** : size(), empty(), max\_size(), ==, !=, <, >, <=, >=의 동작들

- 비교동작은 같은 타입(원소와 정렬기준 둘다 같은 타입)의 컨테이너에 대해서만 제공.

- 다른 타입의 컨테이너를 비교하기 위해서는 반드시 비교 알고리즘을 사용.

**특별한 검색 함수들**

- cout, find, lower\_bound, upper\_bound, equal\_range

- lower\_bound(elem): elem의 값보다 크거나 같은 값을 가지고는 원소의 위치를 반환.

- equal\_range(elem): 정렬된 상태를 깨트리지 않고 elem이 삽입될 수 있는 첫 번째 위치와 마지막 위치 반환.

**할당** : c1=c2 c1.swap(c2) swap(c1,c2)

- 모든 컨테이너가 제공하는 기본적인 할당 연산자만 제공. 같은 타입이라 사실을 기반으로 함.

**반복자 함수**

- 양방향 반복자. 램덤액세스 반복자를 요구하는 알고리즘에서 사용불가. (예. 정렬알고리즘)

- 반복자 입장에서 컨테이너의 모든 원소들은 상수로 취급.(원소의 값 변경이 되어 순서가 흐트려지는 것을 막기 위해서)

**컨테이너 원소의 삽입과 제거**

- 삽입함수의 반환값이 set과 multiset에 따라 다르다. 이유, set은 중복허락을 하지 않지만 multi는 허용

set 인터페이스 : **pair<iterator,bool>** insert(const value\_type& elem);

iterator insert(iterator pos\_hint, const value\_type& elem);

multiset 인터페이스 : iterator insert(const value\_type& elem);

iterator insert(iterator pos\_hint, const value\_type& elem);

1. pair 클래스의 second 멤버는 삽입 연산의 성공여부를 반환한다.

2. pair 클래스의 first 멤버는 새롭게 삽입된 원소의 위치를 반환한다.

만약 pair의 second 멤버가 fail이면, pair클래스의 first 멤버는 기존에 존재하고 있던 원소의 위치를 반환.

**6.6 Map 과 Multimap**

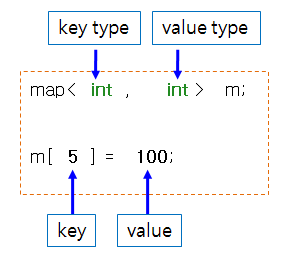
- 연관 컨테이너. 노드기반컨테이너. 균형 2진트리로 구현.

- key/value를 하나의 쌍으로 취급하는 원소를 관리하는 컨테이너.

- 제공된 정렬 기준에 따라서 자신의 원소를 자동적으로 정렬하여 관리(key 기반 정렬)

- map는 key/value의 쌍으로 데이터를 관리. 연관배열처럼 사용.

- 첫번째 템플릿 인자는 원소의 key 타입이고, 두번째 템플릿 인자는 원소의 value타입



1. key/value 쌍은 반드시 할당 가능해야 하며, 복사 가능해야 한다.

2. key는 반드시 정렬 기준에 따라서 비교될 수 있어야 한다.

**능력**

원소의 key를 직접 수정할 수 없다. 수정하고 싶은 key를 erase로 제거하고 새로운 key와 value를 새로운 원소로 삽입해야 한다.

예) pos->first = “hello”; // 컴파일 타임시에 에러 (key는 상수)

예) pos->second = 13.5; // currect , value 값은 변경가능

**동작** - 정렬기준

1. 템플릿의 인자로 넣는 방식

예) std::map<float, std::string, std::greater<float> > coll;

2. 생성자의 인자로 넣는 방식: 정렬기준을 런타임시에 처리하고 싶은 경우 또는 같은 데이터 타입에 대해서 다른 정렬 기준이 필요한 경우 유용.

**반복자 함수와 원소 액세스**

- 직접적인 원소로의 액세스를 제공하지 않는다. 액세스를 하기 위해서 반복자를 사용.

- 예외, map은 원소를 직접액세스 하기 위해 []연산자를 제공.

**컨테이너 원소의 삽입과 제거**

**- map에 원소를 전달하는 3가지 방법**

1. value\_type 사용

-암시적인 형 변환을 피하기 위해서 사용자는 value\_type를 사용하여 정확한 타입을 전달해야만 한다. value\_type은 컨테이너 타입에서 타입정의로 제공된다.

예) std::map<std::string,float> coll;

… coll.insert(std::map<std::string,float>::**value\_type**(“otto”,”22.3”));

2. pair<> 사용

예) coll.insert(std::**pair**<std::string, float>(“otto”,22.3)); //암시적인 형 변환 사용

coll.insert(std::**pair**<const std::string, float>(“otto”,22.3)); //암시적인 형 변환 사용안함

3. make\_pair() 사용

예) coll.insert(std::**make\_pair**(“otto”, 22.3)); //인자로 전달되는 값을 가진 pair 객체 생성.

**6.6.3 연관배열처럼 map 사용하기**

- 연관배열의 장점: map에 보다 편리하게 새로운 원소를 삽입할 수 있다는 점 (인덱스로 정수가 아닌 어떠한 타입도 가질수 있음)

예) coll[“otto”] = 7.7;

- 연관배열의 단점

1. 새로운 원소를 실수로 삽입할 수 있다는 것. 예) std::cout << coll[“ottto”]; // key ottto는 없으므로 Error!

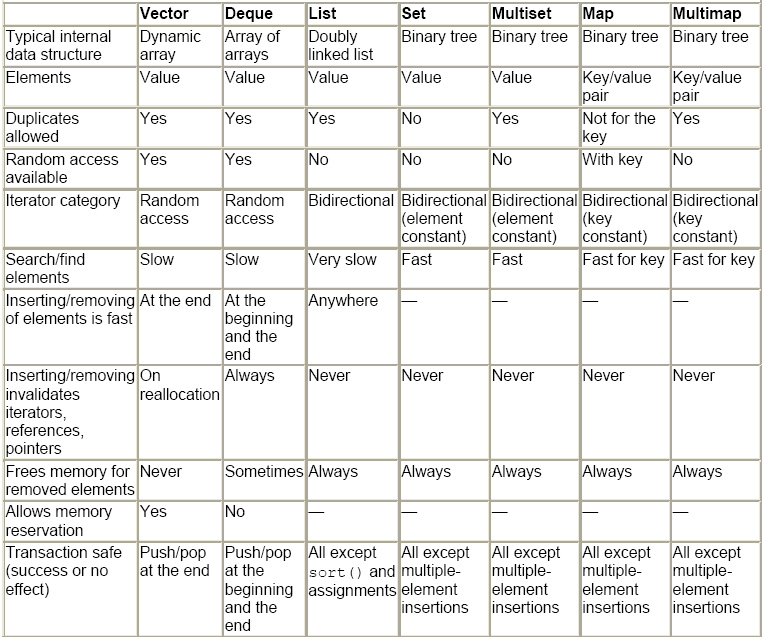
2. 일반적인 삽입방법에 비해서 속도가 느리다. 이유는, value가 먼저 value타입의 디폴트 생성자에 의해서 초기화된 이후, 나중에 사용자가 원하는 값으로 대체되기 때문.

6.8 “레퍼런스 의미론”의 구현

- 일반적으로 컨테이너들은 “값 의미론”을 제공한다. 그러므로 이 컨테이너들은 원소에 대해서 내부적인 복사본을 생성하여 보관하며, 이 원소들을 반환한다.

- 래퍼런스 의미론의 문제점들을 피하기 위해서 스마트 포인터를 구현하여 사용한다.

< 컨테이너의 전반적인 특징 >



▣ **7장 STL반복자**

7.1 반복자의 헤더 파일

- 역방향 반복자와 같은 특별한 반복자에 한해서는 <iterator>헤더에 정의하고 있다.

7.2 반복자 카탈로그

|  |  |  |
| --- | --- | --- |
| *반복자 카테고리* | *기능* | *제공자* |
| 입력 반복자 | 전방향 읽기 | istream |
| 출력 반복자 | 전방향 쓰기 | ostream, inserter |
| 전방향 반복자 | 전방향 읽기, 쓰기 |  |
| 양방향 반복자 | 전방향 역방향 읽기, 쓰기 | list, set, multiset, map, multimap |
| 램덤 액세스 반복자 | 램덤 액세스 | vector, deque, string, array |

**7.2.1 입력 반복자 (Input Iterators)**

- 오로지 원소를 전진 방향으로만 “읽기(read)” 액세스할 수 있는 반복자.

- 한번에 한 개의 원소만 읽어 들일 수 있다.

**7.2.2 출력 반복자 (Output Iterators)**

- 오로지 원소를 전진 방향으로만 “쓰기(write)” 액세스할 수 있는 반복자.

- 사용자가 동일한 범위에 대해서 두번 이상 출력 반복자를 사용하여 순회 할 수 없다.

**7.2.3 전방향 반복자 (Forward Iterators)**

- 입력 반복자인 동시에 출력 반복자인 반복자.

- 입력 반복자의 모든 기능과 출력반복자의 대부분 기능을 가지고 있다.

- 입/출력 반복자와 달리 전방향 반복자는 **같은 컬렉션 안에 존재하는 같은 원소를 한번 이상 참조**할 수 있다.

- **출력반복자**에 대해서는 데이터를 기록할 경우, 시퀀스의 끝을 검사하지 않는 것은 올바른 방법이다.

(출력반복자는 비교동작을 지원하지 않기 때문이다)

예) while(true) { \*pos = foo(); ++pos; } // 검사하는 루틴이 없이 true임

- **전방향 반복자**에 대해서는 사용자는 이 반복자가 유효한지 검사해야 한다.

(반복자가 end()일 경우를 참조할 수 있기 때문)

예) while(pos != coll.end() ) { \*pos = foo(); ++pos; } //end인지 확인

**7.2.4 양방향 반복자 (Bidirectional Iterators)**

: 원소를 역으로 순회할 수 있는 기능이 추가된 전방향 반복자이다.

따라서 모든 기능은 전방향 반복자와 동일하며, 역순으로 순회하기 위한 감소연산자를 제공한다.

**7.2.5 램덤 액세스 반복자(Random access Iterators)**

- 램덤 액세스를 할 수 있는 양방향 반복자이다.

- <, > 연산자를 사용하여 두 반복자를 비교할 수 있다. 예) for(pos=begin(); pos **<** coll.end() -1; pos+=2)

- 양방향 반복자+임의의 위치로 상수 시간에 이동할 수 있도록 산술연산이 가능하다.

- 램덤액세스 반복자는 vector, deque, string, wstring, array, pointer 같은 객체에서만 제공.

7.3 보조 반복자 함수

- advance(), distance() : 램덤 액세스 반복자들만 지원하는 기능을 다른 반복자에도 제공하는 역할.

- iter\_swap() : 두 반복자의 값을 교체하는 역할을 제공

7.3.1 advance()를 사용한 반복자의 위치변경

#include <iterator> void advance ( InputIterator& pos, Dist n) 예) std::advance(pos, 3);

- 입력반복자 pos의 위치를 n원소만큼 증가, 감소시킨다.

따라서 램덤액세스가 아닌 반복자들에 대해 램덤액세스 반복자처럼 동작하게 만든다.

- pos가 양방향 또는 램덤액세스 반복자일 경우 n은 음수일 수 있다.

- Dist는 템플릿 타입이다. <, ++, -- 연산자들이 호출되기 때문에 정수 타입이여만 함.

- 이동시킨 위치가 end()를 넘어가는지 검사를 하지 않는다. 초과할 경우의 동작은 보장되지 않는다.

7.3.2 distance() 함수를 이용한 반복자간의 거리 계산

Dist distance (InputIterator pos1, InputIterator pos2)

- pos1, pos2 사이의 거리를 반환한다.

- pos1, pos2는 같은 컨테이너에 존재하는 원소를 가리키고 있어야 함

- pos1, pos2가 램덤액세스 반복자가 아니라면, pos2는 pos1보다 크거나 같아야 함.

- 반환값은 Dist는 반복자 타입에 따른 difference\_type 타입이다.

- 랜덤 액세스 반복자의 경우 단순히 pos2 - pos1을 반환

- 다른 반복자 카테고리에 대해서는, pos1을 pos2에 도착할 때까지 증가시키고 이 증가횟수를 반환.

따라서 램덤 액세스가 아닐 경우 성능저하 발생

7.3.3 iter\_swap()을 사용하여 반복자의 값 교체

void iter\_swap(ForwardIterator1 pos1, ForwardIterator2 pos2);

- pos1, pos2가 가리키는 값을 교환.

- 반복자가 같은 타입일 필요는 없다. 그러나 가리키는 값이 할당될 수 있어야 한다.

**7.4 반복자 어댑터(Iterator Adapters)**

- 반복자 어댑터를 사용함으로써 역순모드, 삽입모드 등으로 알고리즘의 동작을 변경시킬 수 있다.

**7.4.1 역방향 반복자(Revers Iterators)**

- 자신의 원소들을 역순으로 순회할 수 있도록 하는 역방향 반복자.

- rbegin(): 역방향 반복자의 첫번째 원소를 반환. 정상적인 순서로 본다면 마지막 원소의 위치를 반환.

- rend(): 역방향 반복자의 마지막 원소를 반환. 정상적인 순서로 본다면 첫 번째 원소의 앞 위치를 반환.

반복자와 역방향 반복자(Iterators and Reverse Iterators)

- 일반 반복자를 역방향 반복자로 변경. 반드시 양방향 반복자여야 한다. 논리적인 위치가 변경됨.

예) vector<int>::iterator pos;

vector<int>::**reverse\_iterator rpos**(pos); // 역방향 반복자로 변경

- base()를 사용하여 역방향 반복자를 일반 반복자로 전환하기

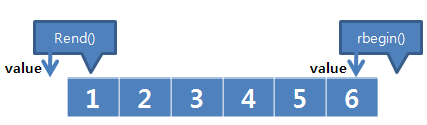
예) list<int>::iterator **pos**;

pos = find(coll.begin(), coll.end(), 5); // 1~9 push\_back(i) 하였다고 가정

list<int>::reverse\_iterator **rpos**(pos); // 반복자를 역방향 반복자로 변경

list<int>::iterator rrpos; **rrpos** = **rpos.base**(); //역방향 반복자를 반복자로 변경

출력결과: pos=5 , rpos=4, rrpos=5 : 반복자가 가리키는 위치는 변하지 않지만 실제로 가리키는 원소 변경.



일반반복자를 역방향반복자 rpos로 변환시 실제 가르치키는 value는 4가 됨.

**7.4.2 삽입 반복자(inserter)**

- 새로운 값을 삽입하는 작업을 수행하기 위한 반복자 어댑터.

|  |  |  |  |
| --- | --- | --- | --- |
| 종 류 | 클래스 | 호출되는 함수 | 생성 |
| 후위 삽입반복자 | back\_insert\_iterator | push\_back(value) | back\_inserter(cont) |
| 전위 삽입반복자 | front\_insert\_iterator | push\_front(value) | front\_inserter(cont) |
| 일반 삽입반복자 | insert\_iterator | insert(pos, value) | inserter(cont, pos) |

**[출처]** [삽입 반복자/스트림반복자](http://blog.naver.com/darkstings/30080095260)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings)

- **후위삽입 반복자**: vector, deque, list, string 컨테이너에서만 제공

예) back\_inserter(coll) = 40; //멤버함수를 호출하여 끝부분에 데이터를 추가

- **전위 삽입 반복자**: deque, list 컨테이너에서만 제공

예) front\_inserter(coll) = 50; //멤버함수를 호출하여 앞부분에 데이터를 추가

- **일반적인 삽입 반복자**

- 두 개의 인자 컨테이너, 삽입될 위치와 함께 초기화됨.

- 삽입된 이후, 새롭게 삽입된 위치를 얻게 됨

예) pos = container.insert(pos, value); ++pos; // 삽입될 위치를 얻음

- insert()에서 반환된 반복자의 위치는 항상 유효하다는 것을 보장.

예) inserter(coll, coll.begin() );

**7.4.3 스트림 반복자**

- 알고리즘의 목적지나 소스에 스트림을 사용하는 것을 허락해 주는 반복자 어댑터임.

**출력 스트림 반복자**

|  |  |
| --- | --- |
| ostream\_iterator<T>(ostream) | ostream에 대해서 출력스트림 반복자를 생성 |
| ostream\_iterator<T>(ostream, delim) | ostream에 대해서 출력스트림 반복자를 생성, 값 사이의 구분자로 delim을 사용 |

- 알고리즘에 출력 스트림 반복자를 사용함으로써 알고리즘의 결과를 스트림에 직접적으로 기록할 수 있다.

**입력 스트림 반복자**

- 스트림으로 원소를 직접 읽어들일 수 있다.

- “끝 스트림 반복자”는 입력 스트림 반복자의 디폴트 생성자와 함께 생성됨.

예) istream\_iterator<int> intReader(cin); //cin으로부터 정수 타입을 읽어들이는 입력 스트림 반복자 생성

istream\_iterator<int> intReaderEOF; // “끝 스트림 반복자” 생성

while ( intReader != intReaderEOF) { cout << “ once : “ << \*intReader << “ once again “ << \*intReader ;

**▣ STL 함수-객체**

8.1 함수-객체의 개념

- 함수-객체는 () 연산자를 정의한 객체이다.

**장점**

1. 기존함수로는 불가능한, 같은 함수에 대해 동시에 다른 상태를 가질 수 있다.

2. 각각의 함수객체는 자신만의 타입을 가지므로 함수객체의 타입을 템플릿의 인자로 제공할 수 있다.

곧, 함수-객체를 가지는 컨테이너를 가질 수 있다는 의미함.

3. 함수 포인터보다 빠르다.

8.1.1 정렬 기준으로서의 함수-객체

: 경우에 따라 컨테이너에 특별한 정렬방식이 필요할 경우 < 연산자에 의존할 수 없다.

따라서 알파벳순서 정렬 등과 같은 특별한 정렬기준으로 함수객체를 사용할 수 있다.

8.1.2 내부 상태를 가지는 함수-객체

- 함수객체는 reference로 전달되기 보다는 value로 전달된다. 그러므로 알고리즘은 함수-객체의 상태를 변경시키지 않는다. 객체의 상태를 변경 시키지 않는 상수성과 임시 객체성의 장점을 가짐.

value로 전달된 함수-객체의 단점은 함수-객체의 상태를 수정할 경우에 얻는 이득을 얻을 수 없다. 알고리즘은 함수-객체를 변경할 수 있다. 그러나 알고리즘은 함수-객체의 내부 복사본을 생성하기 때문에 함수-객체의 최종 상태를 액세스하거나 처리할 수 없다.

🡪 따라서 1. 함수-객체를 레퍼런스로 전달하는 것. 2. for\_each()알고리즘의 반환값을 사용하는 것.

- call-by-reference로 사용시 내부적 변수의 값이 변화하지만, call-by-value로 사용시 객체를 복사하므로 원본에 영향이 없다.

- 함수-객체의 타입이 레퍼런스가 되도록 알고리즘의 호출 부분을 적절하게 수정하여야 한다.

|  |
| --- |
| class **IntSequence** {  private:  int value;  public:  IntSequence (int initialValue) : value(initialValue) {  }  int operator() () {  return value++;  }  };  list<int> coll;  IntSequence seq(1); //1로 시작하는 정수 타입 시퀸스 |

//원소들을 새로 생성하여 삽입(call-by-reference) : 함수객체를 레퍼런스 타입으로 넘김

예) generate\_n <back\_insert\_iterator< list<int> >, int, intSequence&>(back\_inserter(coll), 3, seq);

1~3까지를 레퍼런스 타입으로 넘김. 따라서 다음에는 4부터 시작함!

//원소들을 새로 생성하여 삽입(call-by-value) : 함수객체를 value로 전달

예) generate\_n(back\_inserter(coll), 3, seq); // 4부터 시작함. 4~6

**8.1.3 for\_each()의 반환값**

- for\_each 알고리즘은 함수-객체를 반환하는 특별한 속성을 가지고 있다.

|  |
| --- |
| class MeanValue {  private:  long num; // number of elements  long sum; // sum of all element values  public:  MeanValue () : num(0), sum(0) { }  void operator() (int elem) {  num++; // increment count  sum += elem; // add value  }  double **value** () {  return static\_cast<double>(sum) / static\_cast<double>(num);  }  }; //함수객체를 리턴받아 value()인 멤버함수를 호출하는 예 |

예) MeanValue mv = for\_each(coll.begin(), coll.end(), **MeanValue**() ); //계산된 함수객체를 mv에 할당

cout << "mean value: " << mv.**value()** << endl; //value라는 멤버함수를 호출

8.1.4 조건자와 함수객체

- 조건자는 boolean 값을 반환하는 함수 또는 함수-객체이다.

- 조건자는 호출되는 동안 자신의 상태를 변경해서는 안된다.

- 조건자를 복사한다면, 이 조건자는 반드시 원래의 상태를 변경되지 않는다는 사실을 보장하기 위해서는 사용자가 반드시 ()연산자를 상수 멤버로 선언해야만 한다.

8.2.1 함수 어댑터(Function Adapter)

- 함수-객체를 특정 값 또는 특정 함수와 결합시킨 함수-객체이다.

- 미리 정의된 함수 어댑터

|  |  |
| --- | --- |
| 표현식 | 효과 |
| bind1st(op, value) | op(value, 파라미터) |
| bind2nd(op, value) | op(파라미터, value) |
| not1(op) | !op(파라미터) |
| not2(op) | !op(파라미터1, 파라미터2) |

|  |
| --- |
| vector<int> IntVec;  for(int i=0;i<100;++i) IntVec.push\_back(i);  cout << \*find\_if(IntVec.begin(), IntVec.end(), **bind1st**(greater<int>(), 30)) << endl; //0  cout << \*find\_if(IntVec.begin(), IntVec.end(), **bind2nd**(greater<int>(), 30)) << endl; //31 |

- **bind1st**( greater<int>(), 30); //0 -> true if(30 > \*Iterator)로 검사, 30을 첫번째 파라메터로 고정시켰기때문.

- **bind2nd**( greater<int>(), 30); // 31-> true if( \*Iterator > 30)로 검사, 30을 두번째 파라메터로 고정시켰기때문.

- 함수 어댑터를 함수-객체와 결합하여 사용할 수 있다.

예) pos = find\_if(coll.begin(), coll.end(), **not1**( bind2nd( module<int>(), 2))); //짝수의 값을 가진 원소의 위치…

- 기능적 조합법: 함수어댑터를 이용하여 사용자는 다른 함수-객체들을 매우 강력한 형태로 결합 할 수 있다.

8.2.2 멤버함수를 위한 함수 어댑터

- 컬렉션의 각각의 원소에 대해서 멥버함수를 호출시켜 주는 기능을 가진 함수 어댑터를 제공한다.

- 멤버함수를 알고리즘에 직접적으로 전달할 수 없기 때문에 어댑터가 필요.

- 호출되는 멤버함수들은 반드시 상수멤버함수 이다.

- mem\_fun\_ref(op) : 객체에 대해서 멤버 함수 op()를 호출

- mem\_fun(op) : 객체의 포인터에 대해서 멤버 함수 op()를 호출

예) for\_each(coll.begin(), coll.end(), mem\_fun\_ref(&Person::print)); //인자 없는 멤버함수

- mem\_fun\_ref어댑터는 원소에 대한 함수 호출을 전달되어진 멤버 함수의 호출로 대체 시키는 역할을 함.

- 멤버함수를 알고리즘에 직접전달 할 수 없다.

8.2.3 기존의 함수들을 위한 함수 어댑터

- 기존의 함수를 함수-객체와 같은 형태로 사용할 수 있도록 하는 함수-객체.

- ptr\_fun(op) 파라미터 / 파라미터1, 파라미터2

ex) bool check(int elem); // 체크하는 전역함수

pos = find\_if(coll.begin(), coll.end(), not1(**ptr\_fun**(check))); 파라미터 1개일 때.

pos = find\_if(coll.begin(), coll.end(), bind2nd(**ptr\_fun**(strcmp), “”); 파라미터 2개일 때.

8.2.4 함수 어댑터를 위한 사용자 정의 함수-객체

- 자신만의 함수-객체를 작성하여, 함수 어댑터와 같이 사용할 수 있다.

- 사용자가 작성한 함수-객체들은 함수-객체의 인자와 결과를 위해서 **타입멤버**를 제공해야만 한

다.

8.3 조립 함수-객체

- 기존의 다른 컴포넌트에서 새로운 컨포넌트를 만들고자 할 경우 매우 유용하게 사용할 수 있다.

- 표준화 되어 있지 않음..

- f(g(elem) : elem 을 사용하는 g() 호출의 결과를 조건자 f()의 입력으로 사용한다.

- f(g(elem1, elem2)) : elem1 과 elem2 를 이항 조건자 g()의 인자로 전달하는 형태.

- f(g(elem), h(elem)) : elem 인자를 두 개의 단항조건자 g(), h()에 전달하는 형태

- f(g(elem1), h(elem2)) : elem1, elem2 인자를 다른 단항 조건자 g()와 h()에 전달하는 형태, 이 결과를 이항 조건자 f()의 입력으로 사용한다.

**▣ 9장 STL알고리즘**

9.2.2 알고리즘 분류

1. \_if 접미사

- 동일한 개수의 인자를 받아들이는 경우에 사용된다.

- 단, \_if가 없다면 value가 사용되고, \_if가 존재한다면 value 대신 함수 또는 함수객체가 사용된다.

예) find()는 특정한 값을 원소로 검색하지만 find\_if()는 전달된 함수 또는 함수객체의 기준을 만족시키는 원소를 검색하게 됨.

- 인자의 개수가 다르다면 동일한 이름을 사용한다. ex) min\_element

2. \_copy 접미사

- 원소를 변경할 뿐만 아니라 그 결과를 목적지 범위에 복사하여 전달한다.

예) reverse()는 지정된 범위에서 원소의 순서를 반전하지만, reverse\_copy()는 원소의 반전 결과를 다른 범위에 복사.

**원소를 수정하지 않는 알고리즘**

**-** 원소의 순서나 원소의 값을 변경하지 않는다. **입력반복자, 전방향 반복자**와 함께 동작.

|  |  |
| --- | --- |
| 이 름 | 효 과 |
| for\_each() | 각각의 원소에 대해서 동작을 수행 |
| count() | 원소의 개수를 반환 |
| count\_if() | 기준에 적합한 원소의 개수를 반환 |
| min\_element() | 가장 작은 값을 가지는 원소를 반환 |
| max\_element() | 가장 큰 값을 가지는 원소를 반환 |
| find() | 인자로 전달된 값과 동일한 값을 가지는 첫번째 원소의 위치반환 |
| find\_if() | 기준에 적합한 첫번째 원소의 위치를 반환 |
| search\_n() | 검색조건과 매치되는 원소들이 연속적으로 n번 나타나는 경우를 검색 |
| search() | 첫번째 부분의 범위 검색 |
| find\_end() | 마지막 부분의 범위 검색 |
| find\_first\_of() | 여러 개의 원소, 범위에 대한 검색 |
| adjacent\_find() | 연속된 중복 원소의 검색 |
| equal() | 두 범위가 동일한지 판단 |
| mismatch() | 두 범위에서 다른값의 존재여부 검색 |
| lexicographical\_compare() | 사전식 방법으로 적은지(less) 판단 |

**[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings)

**원소를 수정하는 알고리즘**

- 지정된 범위의 원소들을 직접수정 또는 수정하여 다른 범위에 복사한다. 다른범위로 복사하는 경우는 원본은 변경되지 않는다. 연관컨테이너에 대해서는 자동정렬 유지를 위해 수정 알고리즘을 사용할 수 없다.

|  |  |
| --- | --- |
| for\_each() | 각각의 원소들에 대한 동작을 수행 |
| copy() | 첫번째 원소로 시작하는 범위를 복사 |
| copy\_backward() | 마지막 원소로 시작하는 범위를 복사 |
| transform() | 두 범위의 원소들을 결합하거나 변경 |
| merge() | 두 범위를 병합 |
| swap\_ranges() | 두 범위의 원소들을 교환 |
| fill() | 각각의 원소를 주어진 값으로 교체 |
| fill\_n() | 명시된 n개의 원소를 주어진 값으로 교체 |
| generate() | 각각의 원소를 특정동작의 결과값으로 교체 |
| generate\_n() | 명시된 n개의 원소를 특정동작의 결과값으로 교체 |
| replace() | 특정값을 가지고있던 원소를 다른값으로 교체 |
| replace\_if() | 특정 기준에 부합되는 원소를 다른 값으로 교체 |
| replace\_copy() | 원본의 범위를 복사한 뒤 replace()를 실행 |
| replace\_copy\_if() | 원본의 범위를 복사한 뒤 replace()\_if()를 실행 **[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings) |

- for\_each() : 수정할 값을 인자로 받아들인다(인자는 레퍼런스로 전달)

ex) void square(int& elem) { elem = elem \* elem; }

for\_each(coll.begin(), coll.end(), square);

- transform() : 수정된 값을 반환하는 함수(인자는 value로 전달)

ex) int square(int elem) { return elem \* elem; }

transform(coll.begin(), coll.end(), coll.begin(), square);

**제거 알고리즘**

|  |  |
| --- | --- |
| remove() | 주어진 값과 동일한 원소를 제거 |
| remove\_if() | 주어진 기준을 만족하는 원소를 제거 |
| remove\_copy() | 원본의 범위를 복사한 뒤 remove()를 실행 |
| remove\_copy\_if() | 원본의 범위를 복사한 뒤 remove\_if()를 실행 |
| unique() | 연속 중복된 원소를 제거 |
| unique\_copy() | 원본의 범위를 복사한 뒤 unique()를 실행 **[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings) |

- 수정알고리즘의 특별한 형태. 단 하나의 범위에 존재하는 원소들 뿐만 아니라 다른 범위에 복사된 원소들도 제거할 수 있다.

- 연관 컨테이너의 원소들은 상수처럼 취급됨으로 목적지 범위로 사용될 수 없다.

- 명심 : 제거 알고리즘들은 원소를 논리적으로만 제거한다. 즉, 실제로 제거하는 것이 아니라 제거되지 않은 다음 원소로 덮어쓰기를 하는 것이다. 그러므로 알고리즘 수행한 이후에도 전체의원소 개수는 변경되지 않는다. 대신에, 새로운 논리적인 끝 위치를 반환.

**변경 알고리즘**

|  |  |
| --- | --- |
| reverse() | 원소의 순서를 반전 |
| reverse\_copy() | 문서를 뒤바꾸는 동안 원소를 복사 |
| rotate() | 원소의 순서를 순회 |
| rotate\_copy() | 원소를 순회하는 동일 원소를 복사 |
| next\_permutation() | 원소의 순열 계산 |
| prev\_permutation() | 원소의 순열 계산 |
| random\_shuffle() | 원소를 뒤섞음 |
| partition() | 조건을 만족하는 원소를 앞쪽으로 이동 |
| stable\_partition() | partition()과 동일하나, 원소들의 상대적 순서를 유지 **[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings) |

- 원소의 값을 교체하거나 새롭게 할당하여 원소의 순서를 변경시키는 알고리즘이다.

- 연관 컨테이너의 원소들은 상수처럼 취급됨으로 목적지 범위로 사용될 수 없다.

**정렬 알고리즘**

|  |  |
| --- | --- |
| sort() | 모든 원소를 정렬 |
| stable\_sort() | sort()와 동일하나, 동등 원소의 상대적 위치를 유지 |
| partial\_sort() | 명시된 n 원소까지만 정확히 정렬 |
| partial\_sort\_copy() | copy()와 partial\_sort() 가 결합된 형태 |
| nth\_element() | n번째 원소까지 정렬 |
| partition() | 조건을 만족하는 원소를 앞쪽으로 이동 |
| stable\_partition() | partition()과 동일하나, 원소들의 상대적 순서를 유지 |
| make\_heap() | 지정된 범위의 원소들을 재배치하여 힙으로 구성 |
| pop\_heap() | 힙에 새로운 원소를 제거 |
| sort\_heap() | 정렬( 수행 후 더 이상 힙이 아니다)**[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** 9KM ArtOfRudah |

- 원소의 순서를 변경하기 때문에 변경 알고리즘의 특별한 형태.

- 랜덤 액세스 반복자를 필요로 함.

**정렬된 범위 알고리즘**

**-** 입력으로 들어오는 범위가 정렬기준에 따라서 정렬되어 있는 상태를 요구.

|  |  |
| --- | --- |
| binary\_search() | 특정 원소의 존재여부 판단 |
| includes() | 여러 개의 원소가 존재하는지 판단 |
| lower\_bound() | 값으로 들어온값보다 크거나 같은값을 가지는 원소의 위치반환 |
| upper\_bound() | 값으로 들어온 값보다 큰값을 가지는 원소의 위치반환 |
| equal\_range() | 정렬상태가 깨지지 않는 첫번째, 마지막 위치를 검색 |
| merge() | 두 정렬된 범위를 병합 |
| set\_union() | 두 정렬된 범위의 합집합을 계산 |
| set\_intersection() | 두 정렬된 범위의 교집합을 계산 |
| set\_defference() | 두 정렬된 범위의 차집합을 계산 |
| set\_symmetric\_difference() | 두 정렬된 범위의 대칭 차집합을 계산 |
| inplace\_merge() | 연속적으로 정렬된 범위를 병합 **[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings) |

**수치 알고리즘**

**-** 수치 원소들을 여러가지 방법으로 결합하여 사용할 수 있다.

|  |  |
| --- | --- |
| accumulate() | 모든 원소들의 값을 결합 |
| inner\_product() | 두 시퀀스의 내적을 계산 |
| adjacent\_difference() | 상대적인 값을 절대적인 값으로 변경 |
| partial\_sum() | 절대적인 값과 상대적인 값으로 변경 **[출처]** [알고리즘 Algorithm](http://blog.naver.com/darkstings/30080234915)|**작성자** [9KM ArtOfRudah](http://blog.naver.com/darkstings) |

9.4 for\_each() 알고리즘

UnaryProc **for\_each**(InputIterator beg, InputIterator end, UnaryProc op)

- [beg,end) 범위의 모든 원소에 대해서 op(elem)을 호출

- op(내부적으로 수정된)의 복사본을 반환.

- op는 원소들을 수정할 수도 있다.

- op의 모든 반환 값 들은 무시.

9.5 원소를 수정하지 않는 알고리즘

**-** 원소를 수정할 수 없고 액세스만 할 수 있는 알고리즘.

9.5.1 원소의 카운트(Counting Elements)

difference\_type **count**(InputIterator beg, InputIterator end, const T& value)

: [beg,end) 범위 안에서 value와 동일한 값을 가지는 원소의 개수를 반환

difference\_type **count\_if**(InputIterator beg, InputIterator end, UnaryPredicate op)

**:** [beg,end) 범위 안에서 이항 조건자 op(elem)가 true를 반환하는 원소의 개수를 반환한다.

- 반환값의 타입은 반복자의 “difference\_type” 타입이다.

9.5.2 최소, 최대

ForwardIterator **min\_element** (ForwardIterator beg, ForwardIterator end)

: beg~end범위의 최소값을 가지는 원소의 위치반환

ForwardIterator **min\_element** (ForwardIterator beg, ForwardIterator end, CompFunc op)

: beg~end범위의 최소값을 가지는 원소의 위치반환, 비교조건으로는 op(elem1, elem2)사용

ForwardIterator **max\_element** (ForwardIterator beg, ForwardIterator end)

: beg~end범위의 최대값을 가지는 원소의 위치반환

ForwardIterator **max\_element** (ForwardIterator beg, ForwardIterator end, CompFunc op)

: beg~end범위의 최대값을 가지는 원소의 위치반환, 비교조건으로 op(elem1, elem2) 사용

- op 인자가 없는 경우 < 연산자를 이용하여 비교.

- op는 두 개의 원소를 비교하기 위해서 사용. op(elem1, elem2).

- op인자가 없는경우 < 연산자를 기본으로 사용하며,

op(elem1, elem2) 는 elem1 이 elem2 보다 작을경우 true를 반환해야 한다.

또한, 전달된 인자를 수정해선 안된다.

- 위치를 반환한다는 점. 반환된 값을 출력하기 위해서는 \* 연산자를 사용.

9.5.3 원소의 검색

**검색 조건과 매치되는 첫 번째 원소의 검색**

InputIterator **find**(InputIterator beg, InputIterator end, const T& value)

: 범위 안에서 value와 동일한 값을 가지는 첫번째 원소의 위치를 반환

InputIterator **find\_if**(InputIterator beg, InputIterator end, UnaryPredicate op)

: beg~end범위에서 단항조건자 op(elem)이 true를 반환하는 첫번째 원소의 위치반환

- 매치되는 원소가 없다면 end를 반환

- op는 함수가 호출되는 동안 자신의 상태를 변경해서는 안된다.

- op는 전달 인자를 수정해서는 안된다.

- 연관컨테이너는 동일한 멤버함수를 제공한다.

**검색 조건과 매치되는 원소들이 연속적으로 n번 나타나는 경우를 검색**

ForwardIterator **search\_n** (ForwardIterator beg, ForwardIterator end, Size count, const T& value)

: beg~end 범위에서 value와 동일한 원소가 count만큼 연속되는 곳의 위치 반환

ForwardIterator **search\_n**(ForwardIterator beg, ForwardIterator end, Size count, const T& value, BinaryPredicate

op)

: beg~end 범위에서 이상조건자 op(elem,value)가 true를 반환하는 원소가 count만큼 연속되는 곳의 위치반환.

- 매치되는 원소가 없다면 end를 반환

**첫 번째 부분 범위의 검색**

ForwardIterator1 **search**(ForwardIterator1 beg, ForwardIterator1 end, ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1 **search**(ForwardIterator1 beg, ForwardIterator1 end, ForwardIterator2 searchBeg, ForwardIterator2 searchEnd, BinaryPredicate op)

: 두 번째 범위인 [searchBeg, searchEnd)가 첫 번째 범위인 [beg,end)에 부분범위(subrange)가 되는지 검사한다. 부분 범위가 시작되는 위치를 반환한다.

- 첫 번째 형태의 경우, 모든 원소의 비교는 == 연산자를 호출, 두 번째 형태는 이항 조건자인

op를 호출.

- 매치되는 원소가 없다면 end를 반환.

**마지막 부분 범위 검색**

ForwardIterator1 **find\_end**(ForwardIterator beg, ForwardIterator end, ForwardIterator searchBeg, ForwardIterator searchEnd)

ForwardIterator1 **find\_end**(ForwardIterator beg, ForwardIterator end, ForwardIterator searchBeg, ForwardIterator searchEnd, BinaryPredicate op)

: 두 번째 범위인 [searchBeg, searchEnd)가 첫 번째 범위인 [beg,end)에 부분범위(subrange)가 되

는지 검사한다. 부분위치의 맨 마지막 부분 범위의 위치를 반환.

- 첫 번째 형태의 경우, 모든 원소의 비교는 == 연산자를 호출, 두 번째 형태는 이항 조건자인

op를 호출.

차이) search() 는 첫번째 부분범위의 시작위치를 반환하고, find\_end() 는 마지막 부분범위의 시작위치를 반환한다.

**여러 개의 원소 검색**

ForwardIterator **find\_first\_of**(ForwardIterator1 beg, ForwardIterator1 end, ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

: [beg,end) 범위 안에서 [searchBeg, searchEnd) 범위에 속하는 값들을 검색 한다.

ForwardIterator **find\_first\_of** (ForwardIterator1 beg, ForwardIterator1 end, ForwardIterator2 searchBeg, ForwardIterator2 searchEnd, BinaryPredicate op)

: [beg,end) 범위 안에서 [searchBeg, searchEnd) 범위에 속하는 값들을 검색 한다.

단, op(elem,searchElem) == true 조건을 만족하는 원소를 검색한다.

- find()와 유사하지만 하나의 값이 아니라, 여러 개의 조건을 검색하는 것이다.

- 매치되는 원소가 없다면 end를 반환.

- search(), find\_end()의 경우는 조건범위 전체가 일치해야 하지만, find\_first\_of()는 조건에서 하나의 값이라도 일치한 값으로 간주한다.

- 역방향 반복자를 사용함으로써 검색된 결과의 마지막 위치를 얻을 수 있다.

**연속 중복 원소의 검색**

ForwardIterator **adjacent\_find**(ForwardIterator beg, ForwardIterator end)

ForwardIterator **adjacent\_find**(ForwardIterator beg, ForwardIterator end, BinaryPredicate op)

첫번째 형태는 beg~end 범위에서 첫번째 연속되는 중복원소의 위치를 반환하며,

두번째 형태는 이항조건자 op(elem, nextElem) 가 true를 반환하는 연속적인 첫번째 위치를 반환한다

9.5.4 범위 비교 {비교알고리즘} – 원소를 수정하지 않는 알고리즘의 한 종류

**동일성검사**

bool **equal**(InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg)

bool **equal**(InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg, BinaryPredicate op)

첫번째 형태는 beg~end범위의 원소들과 cmpBeg로 시작하는 범위의 원소들이 같은지 판단한다.

두번째 형태는 비교의 기준으로 이항조건자 op(elem, cmpElem)==true 를 만족하는지 판단한다.

**두 범위에서 다른 값의 존재 여부 검색**

pair<InputIterator1, InputIterator beg>

**mismatch**(InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg)

pair<InputIterator1, InputIterator beg>

**mismatch**(InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg, BinaryPredicate op)

: 상동, 일치하지 않음의 판단은 op(eldm, cmpelem) == false를 이용한다.

첫번째 형태는 beg~end 범위의 원소들을 비교하여 서로 일치하지 않는 첫번째 pair를 반환한다.

두번째 형태는 일치여부의 판단을 op(elem, cmpElem)==false 를 만족하는 첫번째 pair를 반환한다.

모든원소가 일치한다면, 반환되는 pair의 first는 첫번째 범위의 end, second는 두번째 범위의 동일위치이다.

두번째 범위는 첫번째 범위보다 작으면 안되지만, 더 큰 범위를 가지는 것은 무관하다.

**사전식 방법으로 적은지를 판단**

bool **lexicographical\_compare**(InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2, InputIterator2 end2)

bool **lexicographical\_compare**(InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2, InputIterator2 end2, CompFunc op)

: beg1~end1 범위의 원소들이 beg2~end2 범위의 원소들보다 사전식으로 비교했을때 작은지 판단한다.

첫번째 형태는 < 연산자를, 두번째 형태는 이항조건자 op(elem1, elem2)를 비교조건으로 사용한다.

op는 elem1 이 elem2 보다 작을 경우 true를 반환해야 한다.

**사전식비교**는 각각의 원소들에 대하여 elem<elem2 인 경우 곧바로 true를 반환하고,

elem1>elem2 인 경우 곧바로 false를 반환, elem1==elem2 인 경우 다음 pair에 대해 비교한다.

하나의 시퀀스에 더 이상 원소가 없다면 이 시퀀스는 다른 시퀀스보다 작은 원소가 없는 것이다.

따라서 첫번째 시퀀스가 더 이상 원소를 가지고 있지 않다면 결과는 true가 된다.

두 개의 시퀀스 모두 더 이상의 원소가 없다면 두 개의 시퀀스는 동일한 것이다.

따라서 비교 결과는 false가 된다.

9.6 원소를 수정하는 알고리즘 {수정/복사 알고리즘}

1. 시퀀스를 순회하면서 직접적으로 수정.

2. 소스 범위에서 목적지 범위로 복사하는 동안 수정.

9.6.1 원소의 복사

OutputIterator **copy**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)

BidirectionalIterator1 **copy\_backward**(BidirectionalIterator1 sourceBeg, BidirectionalIterator1 sourceEnd, Bidirectional, Iterator2 destEnd)

- 두 알고리즘 모두 sourceBeg~sourceEnd 범위의 모든 원소들을 목적지 범위로 복사하고,

마지막 원소가 복사된 이후의 위치를 반환한다.

목적지 범위는 copy()의 경우 destBeg에서 시작하며, copy\_backward()의 경우 destEnd 로 끝난다.

- copy()는 정방향으로 순회하고, copy\_backward()는 역방향으로 순회한다.

9.6.2 원소의 변경 또는 결합

**두 시퀀스 원소의 결합**

OutputIterator **transform**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg, UnaryFunc op)

OutputIterator **transform**(InputIterator1 source1Beg, InputIterator1 source1End, InputIterator2 source2Beg, OutputIterator destBeg, BinaryFunc op)

첫번째 형태는 소스범위에 있는 원소 목적지 범위로 이동한다. 따라서, 이 형태는 원소의 복사와 수정이 한번에 이루어진다.

두번째 형태는 두개의 소스 시퀀스로부터 원소를 조합하고, 목적지 범위에 그 결과를 기록한다.

9.6.3 원소의 교체(교환)

ForwardIterator2 **swap\_ranges**(ForwardIterator1 beg1, ForwardIterator1 end1, ForwardIterator2 beg2)

: [beg1,end1)의 범위의 원소들을 beg2로 시작하는 원소들과 교체한다.

- 두 번째 범위에서 마지막으로 교체된 원소 이후의 위치를 반환한다.

- 두 컨테이너의 타입이 동일하다면 성능상 이점을 가진 멤버함수 swap()을 사용한다.

9.6.4 새로운 값의 할당

**동일한 값의 할당**

void **fill** (ForwardIterator beg, ForwardIterator end, const T& newValue)

: [beg,end) 범위의 모든 원소에 newValue를 할당한다.

void **fill\_n** (OutputIterator beg, Size num, const T& newVaLue)

: [beg,beg+num) 범위의 원소에 newValue를 할당. 반환값이 없다.

**생성된 값을 할당**

void **generate** (ForwardIterator beg, ForwandIterator end, Func op)

: op()에 의해서 생성된 값을 [beg,end) 범위의 모든 원소에 할당한다.

void **generate\_n** (OututIterator beg, Size num, Func op)

: op()에 의해서 생성된 값을 [beg,beg+num) 범위의 모든 원소를 할당한다. 반환값이 없다.

9.6.5 원소의 교체

**시퀀스 안의 원소들의 값을 교체**

void **replace**(ForwareIterator beg, ForwareIterator end, const T& oldValue, const T& newValue)

: 범위에서 oldValue와 동일한 값을 가지는 원소들을 모두 newValue로 교체

void **replace\_if**(ForwareIterator beg, ForwareIterator end, UnaryPredicate, const T& newValue)

: 범위에서 단항조건자 op(elem)이 true가 되게 하는 원소를 모두 newValue로 교체한다.

**원소의 복사 후 교체**

OutputIterator **replace\_copy** (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg,

const T& oldValue, const T& newValue)

: copy()와 replace()를 합쳐놓은 것이다. [beg~end 범위에서 oldValue와 동일한 값을 가지는 원소들을

destBeg로 시작하는 목적지 범위에 복사한 뒤 newValue로 재할당한다.

OutputIteratrrtoe **replace\_copy\_if**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg,

UnaryPredicate op, const T& newValue)

: copy()와 replace\_if()를 합쳐놓은 것이다 [beg, end) 범위에서 단항 조건자 op(elem)이 true가 되

게 하는 원소들을 destBeg로 시작하는 목적지 범위에 복사한 뒤 newValue로 재할당된다.

9.7 제거 알고리즘

- 원소의 값이나 조건에 따라 범위에서 원소를 제거하는 알고리즘들이다.

- 원소의 개수를 변경할 수 없다. 제거된 원소들을 제거되지 않은 다음의 원소들로 덮어쓰는 것이

다.

- 새로운 논리적 범위의 끝 위치(제거되지 않은 마지막 원소의 다음위치)를 반환한다.

9.7.1 특정 값을 지닌 원소의 제거

**시퀀스 안의 원소들을 제거**

ForwardIterator **remove**(ForwardIterator beg, ForwardIterator end, const T& value)

: 범위안에서 value와 동일한 값을 가지는 원소를 제거.

ForwardIterator **remove\_if**(ForwardIterator beg, ForwardIterator end, UnaryPredicate op)

: 범위안에서 op(elem)이 true를 반환하는 원소들을 제거

- 수정된 시퀀스의 논리적인 마지막 위치를 반환(제거되지 않은 마지막 원소의 다음 위치)

- 제거된 원소들을 제거되지 않은 다음의 원소로 덮어쓴다.

- 제거되지 않은 원소의 순서는 동일하게 유지.

- 원래 컨테이너의 끝 위치 대신에 논리적인 끝 위치를 사용하는 것은 호출자의 몫이다.

- 원소를 수정하기 때문에 연관 컨테이너와는 사용할 수 없다.(erase 멤버함수 제공)

**원소의 제거 결과를 복사하여 전달**

OutputIterator **remove\_copy**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg

const T& value)

:copy와 remove를 결합한 형태. sourceBeg~sourceEnd 범위에서 value와 동일한 값을 가진 원소를 제거하고,

결과를 destBeg로 시작되는 목적지 범위에 복사한다. 원본의 상태는 변화하지 않는다.

OutputIterator **remove\_copy\_if**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg

UnaryPredicate op)

:copy와 remove\_if를 결합한 형태, 소스범위에서 op(elem)이 true를 반환하는 원소를 제거. 결과는

destBeg로 시작되는 목적지 범위에 **복사**되어 기록.

9.7.2 중복된 원소의 제거

**연속 중복된 원소의 제거**

ForwardIterator **unique**(ForwardIterator beg, ForwardIterator end)

: beg~end범위에서 동일한값이 연속되는 원소를 제거한다.

ForwardIterator **unique**(ForwardIterator beg, ForwardIterator end, BinaryPredicate op)

: 이상조건자 op(elem, e)== true를 만족하는 연속되는 원소를 제거한다. 컨테이너가 정렬되어 있다면 중복되는 모든 값이 제거됨.

- 수정된 시퀀스의 논리적인 마지막 위치를 반환(제거되지 않은 마지막 원소의 다음 위치)

- 제거된 원소들을 제거되지 않은 다음의 원소로 덮어쓰며, 제거되지 않은 원소의 순서는 동일하게 유지.

- 원래 컨테이너의 끝 위치 대신에 논리적인 끝 위치를 사용하는 것은 호출자의 몫이다.

- 원소를 수정하기 때문에 연관 컨테이너와는 사용할 수 없다.

- List의 경우 이보다 좋은 unique()멤버함수를 제공한다.

**중복 원소의 제거 결과를 복사하여 전달**

OutputIterator **unique\_copy**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)

OutputIterator **unique\_copy**(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg,

BinaryPredicate op)

- 두 형태 모두 sourceBeg~sourceEnd 범위의 원소에서 연속 중복된 원소를 제거한 결과를 destBeg로 시작되는 목적지 범위에 복사한다. 원본의 상태는 변하지 않는다.

- 두 형태 모두 목적지 범위에 마지막으로 복사된 원소 다음위치를 반환한다.

- copy와 unique알고리즘을 결합한 형태.

- op에 의해서 비교한 후 true이면 elem2를 제거한다.

9.8 변경 알고리즘

- 원소의 순서를 변경하는 알고리즘이다.(값은 변경하지 않는다). 연관컨테이너의 원소는 고정된 순서를 가지

기 때문에 이 알고리즘의 목적지로 사용할 수 없다.

9.8.1 원소의 순서를 반전

void **reverse**(BidirectionalIterator beg, BidirectionalIterator end)

: 범위 안의 원소들의 순서를 뒤집어 놓는다. (반전시킴)

OutputIterator **reverse\_copy**(BidirectionalIterator sourceBeg, BidirectionalIterator sourceEnd,

OutputIterator destBeg)

: sourceBeg~sourceEnd 범위의 원소들의 순서를 반전한 결과를 destBeg로 시작하는 목적지범위에 복사, 목적지의 복사된 마지막 원소의 다음위치를 반환한다.

- List는 멤버함수 reverse를 제공

9.8.2 원소의 회전

시퀀스 안의 원소들을 회전

void **rotate**(ForwardIterator beg, ForwardIterator newBeg, Forwarditerator end)

: beg~end 범위의 원소들을 newBeg가 첫번째 원소가 될때까지 회전시킨다.

: 호출자는 newBeg가 범위 안에서 유효한 위치라는 사실을 보장해야만 한다.

회전된 원소의 결과를 복사하여 전달

OutputIterator **rotate\_copy**(ForwardIterator sourceBeg, ForwardIterator newBeg,

ForwardIterator sourceEnd, OutputIterator destBeg)

: sourceBeg~sourceEnd 범위의 원소들을 newBeg가 첫번째 원소가 될때까지 회전하며, 회전 결과를 destBeg로 시작하는 목적지 범위에 복사한다.

- copy와 rotate가 결합된 형태이다.

9.8.3 원소의 순열 계산

bool **next\_permutation** (BidrectionalIterator beg, BidrectionalIterator end)

bool **next\_permutation** (BidrectionalIterator beg, BidrectionalIterator end, CompFunc op)

: beg~end범위의 원소들의 순서를 내림차순에 따라 변경한다.

bool **prev\_permutation** (BidrectionalIterator beg, BidrectionalIterator end)

bool **prev\_permutation** (BidrectionalIterator beg, BidrectionalIterator end, CompFunc op)

: beg~end범위의 원소들의 순서를 오름차순에 따라 변경한다.

op는 이항조건자 op(elem1, elem2) == true를 만족하는 조건으로 사용한다.

9.8.4 원소를 뒤섞음

void **random\_shuffle** (RandomAccessIterator beg, RandomAccessIterator end)

: beg~end범위의 원소들을 난수 생성기를 이용하여 무작위로 재배치한다.

void **random\_shuffle** (RandomAccessIterator beg, RandomAccessIterator end, RandomFunc& op)

: op를 이용하여 무작위로 재배치한다. op는 반복자의 difference\_type을 인자로 하여 호출된다.

op(max) : op는 0보다 크거나 같고, max보다 작은 난수를 반환해야 함. max자체는 반환 불가.

- op가 비상수 레퍼런스이다. 임시값이나 일반적인 함수들을 전달할 수 없다.

9.8.5 조건을 만족하는 원소를 앞쪽으로 이동하기

BidirectionalIterator **partition** (BidirectionalIterator beg, BidirectionalIterator end, UnaryPredicate op)

BidirectionalIterator **stable\_partition** (BidirectionalIterator beg, BidirectionalIterator end, UnaryPredicate op)

: 두 알고리즘 모두 단항조건자 op(elem)==true를 만족하는 원소를 그렇지않은 원소들보다 앞쪽으로 배치한다.

수행완료후 op()가 false를 반환하는 첫번째 위치를 반환한다.

stable\_partition()은 원소들의 상대적인 순서를 그대로 유지 한다.

9.9 정렬 알고리즘

9.9.1 모든 원소의 정렬

void **sort**(RandomAccessIterator beg, RandomAccessIterator end)

: beg~end 범위의 원소들을 < 연산자를 이용하여 정렬한다.

void **sort**(RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

: 이항조건자를 이용하여 정렬

void **stable\_sort**(RandomAccessIterator beg, RandomAccessIterator end)

: < 연산자를 이용하여 정렬. 상대적인 위치를 유지시켜줌.

void **stable\_sort**(RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

: 이항조건자를 이용하여 정렬

- stable\_sort는 동등 원소의 상대적인 위치를 유지시켜 준다.

- 램덤액세스 반복자를 요구하므로, List에서는 이 알고리즘을 사용할 수 없다.

List의 경우 동일한 기능을 수행하는 멤버함수인 sort()가 제공된다.

9.9.2 부분 정렬

void **partial\_sort** (RandomAccessIterator beg, RandomAccessIterator sortEnd,

RandomAccessIterator end)

: beg~end범위의 원소들 중 beg\_sortEnd까지만 정렬한다. beg~sortEnd범위는 < 연산자에 의해 정렬된다.

void **partial\_sort** (RandomAccessIterator beg, RandomAccessIterator sortEnd, RandomAccessIterator end, BynaryPredicate op)

: op(elem1, elem2)를 이용하여 정렬. beg~sortEnd범위는 op에 의해 정렬된다.

RandomAccessIterator **partial\_sort\_copy** (InputIterator sourceBeg, InputIterator sourceEnd, RandomAccessIterator destBeg, RandomAccessIterator destEnd)

RandomAccessIterator **partial\_sort\_copy** (InputIterator sourceBeg, InputIterator sourceEnd, RandomAccessIterator destBeg, RandomAccessIterator destEnd, BinaryPredicate op)

- copy와 partial\_sort 가 결합된 형태.

- sourceBeg~sourceEnd범위로부터 destBeg~destEnd범위로 원소들을 정렬한 결과를 복사한다. 정렬 후 복사되는 원소들의 개수는 소스와 목적 범위 중 작은쪽의 개수만큼 복사된다.

- 목적지 범위 안에서 마지막으로 복사된 원소 이후의 위치를 반환.

9.9.3 n번째 원소까지 정렬

void **nth\_element** (RandomAccessIterator beg, RandomAccessIterator nth, RandomAccessIterator end)

void **nth\_element** (RandomAccessIterator beg, RandomAccessIterator nth, RandomAccessIterator end, BinaryPredicate op)

: 두형태 모두 nth를 중심으로 왼쪽과 오른쪽으로 분리하여, nth왼편의 원소들이 모두 nth 오른편의 원소들보다 작거나 같도록 정렬한다. 내부적으로 상대적 위치는 보장되지 않음.

- n개의 가장 크거나 작은 원소들만을 필요로 할 경우 사용.

9.9.4 힙 알고리즘

- 힙(heap)이란, 원소를 특별한 방법으로 정렬하는 곳에 사용되는 특별한 시퀀스이다. heap-sort에 의해서 사용. 바이너리 트리로 구현된 특별한 컨테이너이다.

1. 첫 번째 원소의 값은 범위에 속하는 원소들 중에서 가장 큰 값을 가지는 원소.

2. 원소의 추가와 삭제를 로그 시간으로 할 수 있다.

- 힙을 다루기 위한 4가지 알고리즘 제공.

1. make\_heap() : 지정된 범위의 원소들을 재배치하여 힙으로 구성.

void **make\_heap** (RandomAccessIterator beg, RandomAccessIterator end)

void **make\_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

- beg~end 범위의 원소들을 재배치하여 힙으로 구성. op는 이항조건자로서 정렬 기준으로 사용.

- 이 알고리즘은 하나이상의 원소들에 대해 힙 연산을 시작하고자 할 경우에만 사용된다.

2. push\_heap() : 힙에 새로운 원소를 추가

void **push\_head** (RandomAccessIterator beg, RandomAccessIterator end)

void **push\_head** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

- 기존에 존재하는 힙 범위[beg,end-1)에서 마지막 원소를 추가. 따라서 전체 범위인 [beg,end)는 힙이 된다.

3. pop\_heap() : 힙에서부터 다음 원소를 제거한다.

void **pop\_heap** (RandomAccessIterator beg, RandomAccessIterator end)

void **pop\_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

- [beg,end) 범위 안에서 가장 큰 값을 가지는 첫 번째 원소를 마지막 위치로 이동시킨 후,

[beg,end-1) 범위에 남아있는 원소들을 가지고 새로운 힙을 생성.

4. sort\_heap() : 범위의 원소들을 정렬(실행된 이후는 힙이 아니다)

void **sort\_heap** (RandomAccessIterator beg, RandomAccessIterator end)

void **sort\_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

- 범위의 원소들을 정렬.

- 이 알고리즘을 호출한 이후 더 이상 **힙이 아니다.**

9.10 정렬된 범위 알고리즘

- 소스범위가 정렬되어 있다는 가정하에 동작하는 알고리즘이다.

이 알고리즘은 기존의 정렬되어 있지 않은 소스범위를 취하는 알고리즘보다 좋은 성능을 보장함.

램덤액세스 반복자가 아니더라도 사용할 수 있다. 정렬되지 않은 시퀀스에 대한 동작은 결과가 보장되지 않는다. 연관컨테이너의 경우 검색 알고리즘들을 멤버함수로 제공한다.

9.10.1 원소의 검색

**원소의 존재 여부 판단**

bool **binary\_search** (ForwardIterator beg, ForwardIterator end, const T& value)

bool **binary\_search** (ForwardIterator beg, ForwardIterator end, const T& value, BinaryPredicate op)

- 정렬된 범위안에 value와 같은 값이 존재하는지 판단한다.

- 검색한 원소의 위치를 얻기 위해서는 lower\_bound(), upper\_bound(), 또는 equal\_range()를 사용.

- 호출자는 소스 범위가 정렬 기준에 따라 정렬되어 있다는 사실을 보장.

**여러 개의 원소가 존재하는지 판단**

bool **includes** (InputIterator1 beg, InputIterator end, InputIterator2 searchBeg, InputIterator2 searchEnd)

bool **includes** (InputIterator1 beg, InputIterator end, InputIterator2 searchBeg, InputIterator2 searchEnd,

BinaryPredicate op)

- [beg,end)가 정렬된 범위인 [searchBeg, searchEnd)의 모든 원소를 포함하고 있는지 판단한다.

- true이면 다 포함하고 있다는 뜻.

**정렬 상태가 깨지지 않는 첫 번째 또는 마지막 위치의 검색**

ForwardIterator **lower\_bound** (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator **lower\_bound** (ForwardIterator beg, ForwardIterator end, const T& value, BinaryPredicate op)

: value로 들어온 값보다 크거나 같은 값을 가지는 원소의 위치.

ForwardIterator **upper\_bound** (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator **upper\_bound** (ForwardIterator beg, ForwardIterator end, const T& value, BinaryPredicate op)

: value로 들어온 값보다 큰 값을 가지는 원소의 위치

- 만족하는 원소가 없으면 end를 반환.

- 두 가지 조건의 반환값을 동시에 얻고 싶다면, equal\_range()를 사용.

pair<ForwardIterator, ForwardIterator>

**equal\_range** (ForwardIterator beg, ForwardIterator end, const T& value)

pair<ForwardIterator, ForwardIterator>

**equal\_range** (ForwardIterator beg, ForwardIterator end, const T& value, BinaryPredicate op)

: lower\_bound(), upper\_bound()의 결과를 pair로 묶어 반환한다.

pair의 first는 lower\_bound(), second는 upper\_bound() 의 결과와 동일하다.

- make\_pair(lower\_bound(…), upper\_bound(…)) 와 동일.

9.10.2 원소의 병합

- 두 범위의 원소들을 하나로 합친다. 두 범위를 합치거나 합집합, 교집합, 차집합 등을 구할 수

있다.

정렬된 두 범위 합치기

OutputIterator **merge** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg)

OutputIterator **merge** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg, BinaryPredicate op)

- 정렬된 소스범위의 원소를 하나로 합치고, destBeg로 시작하는 목적지 범위에 기록한다.

- 소스 범위는 수정되지 않으며 목적지 범위에 복사된 모든 원소들은 정렬된 순서를 가진다.

**정렬된 두 범위의 합집합 계산**

OutputIterator **set\_union** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg)

OutputIterator **set\_union** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg, BinaryPredicate op)

: source1, source2의 원소들의 합집합을 destBeg로 시작하는 목적지 범위에 복사한다. 복사된 목적지 범위의 모든 원소들은 정렬된 순서를 가진다. (중복시, 더 많은 개수를 보유한쪽을 따라간다)

**정렬된 두 범위의 교집합 계산**

OutputIterator **set\_intersection** (InputIterator source1Beg, InputIterator source1End,

InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg)

OutputIterator **set\_intersection** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg, BinaryPredicate op)

: source1, source2의 원소들의 교집합을 destBeg로 시작하는 목적지 범위에 복사한다. 복사된 목적지 범위의 모든 원소들은 정렬된 순서를 가진다. (중복시, 더 작은 개수를 보유한쪽을 따라간다)

**정렬된 두 범위의 차집합 계산**

OutputIterator **set\_difference** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg)

OutputIterator **set\_ difference** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg, BinaryPredicate op)

**-** 첫 번째 범위에만 속하고, 두 번째 범위에는 속하지 않는 원소들로 구성.

OutputIterator **set\_symmetric\_difference** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg)

OutputIterator **set\_symmetric\_difference** (InputIterator source1Beg, InputIterator source1End, InputIterator source2Beg, InputIterator source2End, OutputIterator destBeg, BinaryPredicate op)

- 대칭차집합: 첫 번째 범위에만 속해 있고 두 번째 범위에는 속하지 않는 원소들과 첫 번째 구간에는 속해 있지 않고 두 번째 구간에만 속하는 원소들로 구성된 집합을 의미.

**연속적으로 정렬된 범위의 병합** {내부적인 2개의 정렬범위를 하나로 병합}

void **inplace\_merge** (BidirectionalIterator beg1, BidirectionalIterator end1beg2, BidirectionalIterator end2)

void **inplace\_merge** (BidirectionalIterator beg1, BidirectionalIterator end1beg2, BidirectionalIterator end2, BinaryPredicate op)

- 두 개의 정렬 범위인 [beg1,end1beg2)와 [end1beg2,end2)를 합쳐서 그 결과를 [beg1,end2)에 놓는다.

9.11 수치 알고리즘

- <numeric>헤더 파일을 추가.

- 수치값 뿐만 아니라 다른 타입도 처리가능.

9.11.1 결과값 처리

**한 시퀀스의 결과 계산**

T **accumulate** (InputIterator beg, InputIterator end, T initValue)

: 범위의 각각의 원소에 대해서 initValue = initValue + elem을 계산, initValue의 최종 합을 반환

T **accumulate** (InputIterator beg, InputIterator end, T initValue, BinaryFunc op)

: 범위의 각각의 원소에 대해서 initValue = op(initValue, elem)를 호출하여 계산, initValue의 최종합을 반환

**두 시퀀스의 내적 계산**

T **inner\_product** (InputIterator beg1, InputIterator end1, InputIterator beg2, T initValue)

: [beg,end)범위의 각각의 원소와 beg2로 시작하는 두 번째 범위에 대해서

initValue = initValue + elem1 \* elem2를 계산, initValue의 최종 내적값을 반환.

T **inner\_product** (InputIterator beg1, InputIterator end1, InputIterator beg2, T initValue, BinaryFunc op1, BinaryFunc op2)

: intiValue = op1(initValue, op2(elem1,elem2))을 계산, initValue의 최종 내적값을 반환

- 첫 번째 범위가 비어있다면, 두 형태 모두 initValue를 반환

9.11.2 절대적인 값과 상대적인 값으로의 변경

**상대적인 값을 절대적인 값으로의 변경**

OutputIterator **partial\_sum** (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)

: 소스범위의 각 원소에 대해서 부분합을 계산하여 destBeg로 시작하는 목적지 범위에 복사한다.

OutputIterator **partial\_sum** (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg, BinaryFunc op)

: op를 호출하여 계산 후 목적지 범위에 결과를 기록

- 마지막으로 복사된 목적지 원소의 다음위치를 반환한다.

**절대적인 값에서 상대적인 값으로의 변경**

OutputIterator **adjacent\_difference** (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)

: 범위의 각원소들에 대한 차이를 계산한 결과를 destBeg로 시작하는 목적지 범위에 복사한다.

OutputIterator **adjacent\_difference** (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg, BinaryFunc op)

: 범위에 대해 op를 호출한 결과를 destBeg로 시작하는 목적지 범위에 복사한다. 소스와 목적지 범위는 동일할 수 있음.

호출자는 목적지 범위가 충분하지 않을 경우, 삽입반복자를 사용하는 것을 보장한다.

▣ 과제

1. pos 범위가 end() 범위를 넘을 경우 무한루프나 예상치 못한 결과를 가져온다. (연산자 != , < 사용)

- 루프의 종료조건으로 제공된 < 연산자는 램덤 액세스 반복자만 사용가능

- 만약 end()의 범위를 벗어난 (end() + 2) 루프의 경우 문제가 될 수 있다.

for (pos = coll.begin(); pos < coll.end()-1; pos += 2)

{

cout << \*pos << ' ';

}

// 만약 pos가 end 범위를 넘어서게 되면 ? 어떤결과를 가져올지 모른다.(혹은 무한루프에 빠짐)

pos = coll.end()-2; //8

for (; pos != coll.end()-1; pos += 2)

{

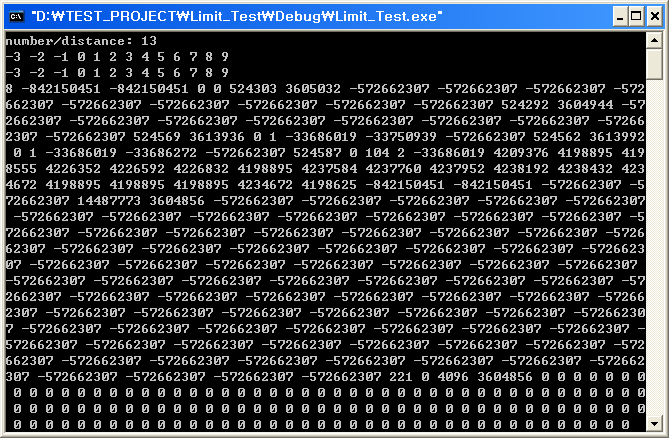
cout << \*pos << ' '; //8 출력후 coll.end() -1과 같지 않으므로 +=2를 수행하여 end()의 범위를 넘음!

}

[ -3(begin()) -2 -1 0 1 2 3 4 5 6 7 8 9 end() ] 순으로 정렬됨.

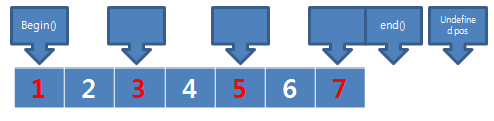
8 출력후 9 다음의 end를 출력하므로 예상치 못한 결과값을 출력한다.

이후 조건과 일치 하지 않아 무한 루프에 빠지게 된다.



< 코드 >

|  |
| --- |
| #include <vector>  #include <iostream>  using namespace std;  int main()  {  vector<int> coll;  // insert elements from -3 to 9  for (int i=-3; i<=9; ++i) {  coll.push\_back (i);  }  /\* print number of elements by processing the distance between beginning and end  \* - NOTE: uses operator - for iterators \*/  cout << "number/distance: " << coll.end()-coll.begin() << endl;  /\* print all elements  \* - NOTE: uses operator < instead of operator != \*/  vector<int>::iterator pos;  for (pos=coll.begin(); pos<coll.end(); ++pos) {  cout << \*pos << ' ';  }  cout << endl;  /\* print all elements  \* - NOTE: uses operator [] instead of operator \* \*/  for (int j=0; j<coll.size(); ++j) {  cout << coll.begin()[j] << ' ';  }  cout << endl;  /\* print every second element  \* - NOTE: uses operator += \*/  **for (pos = coll.begin(); pos < coll.end()-1; pos += 2) {**  cout << \*pos << ' ';  }  cout << endl;  return 0;  } |



pos +=2 로 end()의 범위를 벗어나게 된다면 알수없는 포지션의 값을 가져오며 조건문으로 제어할 수 없다.

2. 각 컨테이너의 쓰이는 예시

1. vector : 한쪽에서 입출력이 일어나는 예로 기본적으로 씀. 예) 원소표

2. deque : 대부분의 삽입, 삭제가 앞, 뒤로 입출력이 많은 처리.

3. list: 링크드리스트 자료구조로 상품 데이타의 관리와 삽입,삭제, 검색 등을 관리 예)로고쌓기(뒤에 쌓기)

4. set: 자동정렬, 유일한값이 존재하는 특성을 이용한 회원관리

5. multiset: 자동정렬된 원소를 이용한 회원관리, 도서관리

6. map: key/value 의 구조를 이용한 연관 배열로 씀

7. multimap: key/value의 구조의 중복 key를 이용한 사전(dic)