**STL 기초 – 변경 알고리즘3 (요소 제거)**

**요소제거 : 요소를 제거하는 함수는 조건에 맞는 요소를 찾아 제거하는 시늉을 함(실제로 요소를 제거하지 못함 왜냐하면 일반성을 가진 알고리즘이 컨테이너를 조작 할 권한이 없기 때문)**

FwdIt remove(FwdIt first, FwdIt last, const Type& val);

FwdIt remove\_if(FwdIt first, FwdIt last, UniPred F);

OutIt remove\_copy(FwdIt first, FwdIt last, OutIt result, const Type& val);

OutIt remove\_copy\_if(FwdIt first, FwdIt last, OutIt result, UniPred F);

* remove 함수는 작업 대상이 되는 반복자 구간과 제거할 값을 인수로 전달받아 first ~ last 사이의 값을 제거한다
* remove는 실제로 요소를 제거하지는 않으며 제거 대상이 아닌 요소들을 골라 구간의 앞쪽으로 이동시키고 남은 요소의 시작을 가리키는 반복자를 리턴한다
* 제거 대상이 아닌 요소의 원래 순서는 유지되므로 안정성이 있다
* 요소가 실제로 제거되지 않고 위치만 바뀌므로 remove 호출 후에도 컨테이너의 크기는 변하지 않는다
* 요소를 실제로 제거하려면 컨테이너의 erase 멤버 함수를 호출해야 한다

remove 예제

#include <iostream>

#include <vector>

#include <algorithm>

#include <iterator>

using namespace std;

template<typename C> void dump(const char \*desc, C c) { cout.width(12);cout << left << desc << "==> ";

copy(c.begin(),c.end(),ostream\_iterator<typename C::value\_type>(cout," ")); cout << endl; }

int main()

{

int ari[]={3,1,4,1,5,9,2,6,5};

vector<int> vi(&ari[0],&ari[sizeof(ari)/sizeof(ari[0])]);

vector<int>::iterator it;

dump("원본",vi);

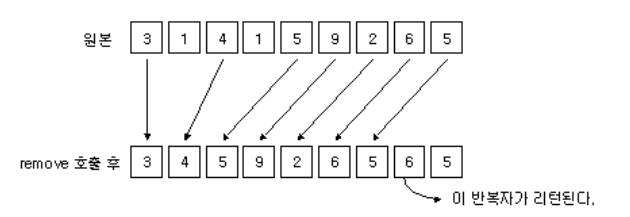
it=remove(vi.begin(),vi.end(),1);

dump("remove",vi);

vi.erase(it,vi.end());

dump("erase",vi);

}

****

🡺remove는 구간의 처음부터 순회하면서 제거 대상이 아닌 요소를 앞쪽으로 차례대로 복사한다. 선두값인 3은 제거 대상인 1이 아니므로 그대로 남는다. 다음값 1은 제거 대상이므로 복사되지 않는다

\*\*\*remove는 왜 이렇게 복잡한 과정을 거쳐서 요소를 삭제하도록 되어 있을까?

* 그냥 1을 삭제해 버리고 1이 아닌 것들만 남겨둔다면 굳이 erase를 호출할 필요가 없지 않은가? 이렇게 되어 있는 이유는 어떤 장점이 있어서 그런 것이 아니라 일반성을 가진 알고리즘이 할 수 있는 동작이 제한되어 있기 때문이다. remove는 지금 자신이 검색하고 있는 컨테이너가 벡터라는 것을 모르며 따라서 실제로 요소를 삭제하기 위해 어떻게 컨테이너를 조작해야 하는지도 알지 못한다.
* 실제 삭제는 해당 컨테이너의 erase 멤버 함수가 처리해야 한다. 반면 멤버 함수는 컨테이너별로 고유하며 컨테이너의 구조를 알고 있으므로 지정한 요소를 삭제할 수 있다. 리스트의 remove 멤버 함수는 멤버이므로 검색과 동시에 삭제가 가능하다. 좀 불편하기는 하지만 이런 불편함의 대가로 일반성이 얻어진다. 만약 삭제 대상을 선정하고 실제로 삭제하는 것이 번거롭다면 새로운 컨테이너를 생성하는 remove\_copy 함수를 사용한다

remove\_copy 예제

#include <iostream>

#include <vector>

#include <algorithm>

#include <iterator>

using namespace std;

template<typename C> void dump(const char \*desc, C c) { cout.width(12);cout << left << desc << "==> ";

copy(c.begin(),c.end(),ostream\_iterator<typename C::value\_type>(cout," ")); cout << endl; }

int main()

{

int ari[]={3,1,4,1,5,9,2,6,5};

vector<int> vi(&ari[0],&ari[sizeof(ari)/sizeof(ari[0])]);

vector<int> vi2;

remove\_copy(vi.begin(),vi.end(),back\_inserter(vi2),1);

dump("vi",vi);

dump("vi2",vi2);

}

* remove\_copy는 제거 대상을 제외한 요소들을 새로운 반복자 구간에 대입한다. \_copy가 뒤에 붙은 함수는 항상 작업 결과를 다른 구간으로 복사하는 함수이다.
* remove\_copy 함수와 삽입 반복자를 사용하면 빈 벡터에 원하는 요소만 모을 수 있다.

\*remove\_copy 함수는 이 밑에 코드와 사실상 동일 코드

vi2=vi; // 또는 copy(vi.begin(),vi.end(),back\_inserter(vi2));

vi2.erase(remove(vi2.begin(),vi2.end(),1),vi2.end());

* 대입이나 copy 함수로 일단 똑같은 컨테이너를 만든 후 remove로 특정 요소를 제거한 후 erase로 짤라내면 결국 vi2에는 1이 아닌 요소만 남는다. 그러나 보다시피 이 방법은 대입이라는 비싼 연산을 거쳐야 하므로 느리다. 복사 함수는 새로운 컨테이너를 만들면서 제거를 겸할 수 있어 복사 후 제거보다 훨씬 더 빠르다. 이런 이유로 STL은 알고리즘을 적용하면서 그 결과를 다른 컨테이너에 작성하는 \_copy 류의 함수를 제공하는 것이다. 다음 함수도 일종의 제거 함수인데 동작하는 방식은 remove와 유사하다.

FwdIt unique(FwdIt first, FwdIt last [,BinPred F]);

OutIt unique(FwdIt first, FwdIt last, OutIt result [,BinPred F]);

* unique 함수는 반복자 구간에서 연속된 중복 요소를 제거한다. 이때 중복되었다는 조건의 디폴트는 == 이지만 이항 조건자 F로 사용자가 중복 조건을 지정할 수도 있다. remove와 마찬가지로 실제로 요소를 삭제하지는 않으며 중복되지 않은 요소들만 앞쪽으로 이동시킨다

unique 예제

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

char str[]="abcccddefggghi";

char \*p;

puts(str);

p=unique(&str[0],&str[strlen(str)]);

\*p=0;

puts(str);

}

* 주의! unique는 구간내의 중복된 것을 모두 제거하는 것이 아니라 인접된 중복 요소만 제거한다
* 멀리 떨어져있는 중복값은 제거되지 않는다
* 만약 중복된 모든 요소를 완전히 제거하고 싶다면 정렬을 먼저 한 후 unique를 적용해야 한다
* unique\_copy는 중복되지 않은 요소를 다른 컨테이너로 복사한다