STL 컨테이너

김성익(noerror@hitel.net)
2005.04.07

개요

- Standard Template Library
- 적절한 컨테이너 선택
- Template / namespace

- 순차 컨테이너 *vector, list, deque*
- 연관 컨테이너 *map, set, multimap, multiset*

반복자

- STL 컨테이너의 공통적인 element 접근 방법
- 순회
- 삭제, 삽입, 검색

순차 컨테이너(1)

- Element 를 리스트 앞이나 뒤에 삽입하며, 순서가 그대로 유지되는 컨테이너
- iterator를 이용한 공통적인 순회 방식

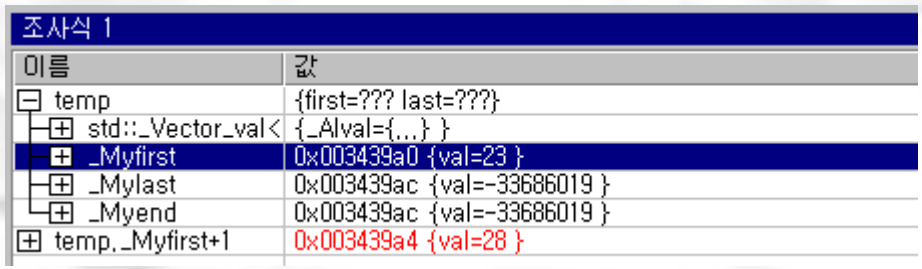
```
#include <stdio.h>
#include <vector>

using namespace std;

void main()
{
    vector <int> temp;
    vector <int> :: iterator i1;
    vector <int> :: reverse_iterator i2;

    temp.push_back(22);
    temp.push_back(23);

    for(i1=temp.begin(); i1!=temp.end(); i1++)
        printf("%d ", (*i1));

    for(i2=temp.rbegin(); i2!=temp.rend(); i2++)
        printf("%d ", (*i2));
}
```

순차 컨테이너(2)

- vector
- list
- deque : double-ended queue

vector 특징

- 내부적으로 배열과 거의 동일
  동적인 배열??
- 접근이 빠르다
- 디버깅시 내부 직접 접근하기 좋다



vector 삽입/삭제

- 삽입 (리스트의 마지막에): push_back
  배열의 크기가 부족한 경우 배열을 늘린 후 element 를 복사
- 삭제 (반복자 이용)
  vector는 삭제 후 계속 순회하는 것도 가능하지만 다른 컨테이너는 유효하지 않다

```
vector <int> temp;
vector <int> :: iterator ii;

for(ii=temp.begin(); ii!=temp.end(); ii++)
{
    if ((*ii) == 50)
    {
        temp.erase(ii);
        break;
    }
}
```

vector 삽입/삭제

- 삭제 비용이 크다

  삭제 후 뒷부분의 element를 하나씩 앞당겨 복사한다

```
class   _dummy
{
public :
    _dummy(int v) { val = v;}

    void Do() { printf("%d₩n", val); }

    _dummy * operator = (const _dummy & dum) {
        val = dum.val;
        printf("copy₩n");
        return this;
    }
private :
    int val;
} ;

void main()
{
    vector <_dummy> temp;
    vector <_dummy> ::iterator ii;

    temp.push_back(_dummy(23));
    temp.push_back(_dummy(28));
    temp.push_back(_dummy(29));

    temp.erase(temp.begin());

    for(ii=temp.begin(); ii!=temp.end(); ii++)
        (*ii).Do();
}
```

vector 접근

- 반복자를 이용한 접근
- [] 오퍼레이터를 이용한 랜덤 접근

```
vector <int> temp;
vector <int> :: iterator ii;
unsigned int i;

for(ii=temp.begin(); ii!=temp.end(); ii++)
    printf("%d₩n", (*ii));

for(i=0; i<temp.size(); i++)
    printf("%d₩n", temp[i]);
```

vector 소팅

- 가벼운 소팅

```
#include <vector>
#include <stdio.h>
#include <algorithm>

struct _item {
    _item(int k) { a = k; }
    int a;
} ;

bool _sort(const _item a, const _item b)
{
    return a.a < b.a ? true : false;
}

using namespace std;

void main()
{
    vector <_item> temp;
    unsigned int i;

    temp.push_back(_item(10));
    temp.push_back(_item(5));
    temp.push_back(_item(8));

    std::sort(temp.begin(), temp.end(), _sort);

    for(i=0; i<temp.size(); i++)
        printf("%d ", temp[i].a);
}
```

deque 특징

- 리스트 앞/뒤에 삽입 가능
- 일부 배열의 특성을 비교적 유지
  내부 블록은 배열처럼 연속된 메모리로 구성
- 랜덤 억세스 가능

```
#include <deque>

using namespace std;

void main()
{
    deque <int> temp;
    deque <int> :: iterator ii;

    temp.push_back(23);
    temp.push_back(24);
    temp.push_front(22);
    temp.push_back(25);

    for(ii=temp.begin(); ii!=temp.end(); ii++)
        printf("%d ", (*ii));
}
```

list 특징

- 내부적으로 링크드 리스트 구조
- vector에 비해서 삽입/삭제 비용이 저렴
- 리스트에 앞/뒤에 삽입 가능
  중간 삽입 비용도 저렴
- 랜덤 억세스 불가능

```
#include <list>

using namespace std;

void main()
{
    list <int> temp;
    list <int> :: iterator ii;

    temp.push_back(23);
    temp.push_back(24);
    temp.push_front(22);

    for(ii=temp.begin(); ii!=temp.end(); ii++)
        printf("%d ", (*ii));
}
```

연관 컨테이너

- 내부적인 순서와 상관없이 배열되는 구조
  내부 리스트 구성 알고리즘에 대한 정해진 규칙은 없다
- 주로 검색이 많은 자료 구조형에 사용

map 특징

- 검색이 빠르다
- element는 pair 구조로 두 개의 값을 가진다 (키 값, 데이터 값)
- 중복된 키 값은 가질 수 없다

map사용(1)

- 반복자, 혹은 [] 연산자를 이용해서 접근 가능하다

```
#include <map>

using namespace std;

void main()
{
    map <int, int> temp;
    map <int, int> ::iterator ii;

    temp[25] = 3;
    temp.insert(pair <int, int> (29, 23));

    printf("temp[25] = %d₩n", temp[25]);

    ii = temp.find(29);
    if (ii != temp.end())
        printf("temp[29] = %d₩n", (*ii).second);
}
```

map사용(2)

- 키 값을 스트링을 사용
  빈번하게 사용되는 예

```
#include <map>
#include <string>

using namespace std;

void main()
{
    map <string, int> temp;

    temp["test2"] = 3;

    printf("temp[25] = %d₩n", temp["test2"]);
}
```

map 비교자 활용

- string대신 문자열 포인터를 키로 사용
  키 값이 안 바뀌고, 포인터가 항상 유효 해야 함

```
#include <map>
#include <string.h>

using namespace std;

struct _element {
    char key[16];
    int value;
    _element(char * k) { strcpy(key, k); }
} ;

template <class _Tx> struct less_str : binary_function <_Tx, _Tx, bool>
{
public:
   bool operator() (const _Tx &a, const _Tx &b) const { return strcmp(a, b) < 0 ? true : false; }
} ;

void main()
{
    map <const char *, _element *, less_str <const char*> > temp;
    map <const char *, _element *, less_str <const char*> > ::iterator ii;
    _element * e;

    e = new _element("test");
    e->value = 55;
    temp[e->key] = e;

    ii = temp.find("test");
    if (ii != temp.end())
        printf("temp[%s] = %d₩n", (*ii).first, (*ii).second->value);
}
```

multimap 특징

- map과 거의 동일한 특성
- 같은 키를 가지는 여러 element 존재
- []연산자 사용 불가

```
#include <map>

using namespace std;

void main()
{
    multimap <int, int> temp;
    multimap <int, int> ::iterator ii, last;

    temp.insert(pair <int, int> (8, 3));
    temp.insert(pair <int, int> (9, 23));
    temp.insert(pair <int, int> (8, 23));

    ii = temp.find(8);
    if (ii != temp.end())
    {
        last = temp.upper_bound(8);

        for(; ii != last; ii++)
            printf("temp[8] = %d₩n", (*ii).second);
    }
}
```

set 특징

- map과 비슷하지만 키 값만 존재
- 단순히 존재하느냐 아니냐 검색에 사용

```
#include <set>

using namespace std;

void main()
{
    set <int> temp;

    temp.insert(323);
    temp.insert(33);

    if (temp.find(33) != temp.end())
        printf("temp[33] is set");
}
```

- multiset은 여러 개의 키 값의 element 존재가능한 자료형

기타

- Thread unsafe 하지만, 그렇기 때문에 오버헤드가 없다
- 메모리 할당자
- reserve, resize
- find_if, for_each,...

- STL은 만능입니까 ?

참고

- Standard Template Library Programmer's Guide
  http://www.sgi.com/tech/stl/
- An introduction of STL for beginners
  http://www.mindcracker.com/mindcracker/c_cafe/stl/stlt1.asp